QUANTITATIVE ISSUE

2026. 3. 17

MACRO팀

김동영, CFA Quant Analyst
dy76.kim@samsung.com

리서치센터 리포트
바로가기

퀀트 모델링 A to Z

(11) Attention/Transformer의 이해, 2부

- Transformer 아키텍처의 내부 알고리즘 및 구조에 대하여 상세히 설명하고, 투자 관점에서의 모델 활용법을 소개함

WHAT'S THE STORY?

세상의 변화를 주도하는 대부분의 LLM 모델의 뿌리에는 "Attention Is All You Need" 논문의 Transformer 모델이 존재한다. 이 Transformer의 변형 모델이, 여러 산업의 다양한 응용 분야에서 널리 활용되고 있다.

이번 "Attention/Transformer의 이해, 2부"에서는, 본격적으로 Transformer 아키텍처의 세부 구조에 대해서 상세히 설명한다. 그 주요 대상은 Multi-Head Attention 및 Self-Attention, Position-wise Feed-Forward Networks, Sinusoidal Positional Encoding 등이다.

이후에는 keras 등을 기반으로 한 실제 Transformer 모델 사용법을 정리한다. 마지막에는, 논문 리뷰를 통해 실제 금융시장 분석에 활용하는 방법에 대해서도 점검한다.

Attention formula

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

자료: "Attention Is All You Need" 논문

삼성증권

REPORT
CONTENTS

Ⅰ 서론 03p

Ⅱ Transformer 아키텍처 04p

0. 임베딩 벡터
1. Residual Connection, Layer Normalization
2. Scaled Dot-Product Attention
3. Multi-Head Attention
4. Self-Attention
5. Masked Attention
6. Position-wise Feed-Forward Networks
7. Sinusoidal Positional Encoding
8. 정리

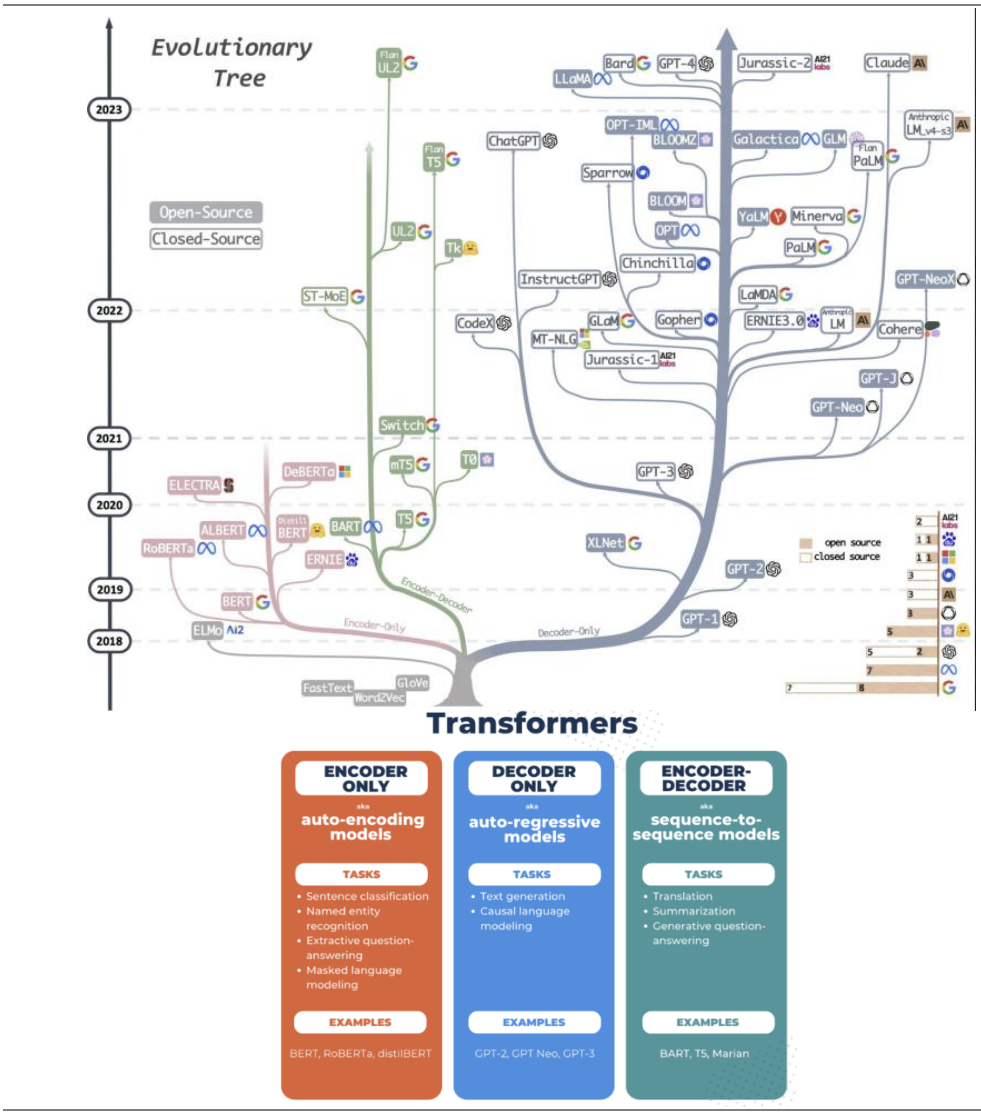
Ⅲ Transformer 사용법 27p

Ⅳ Transformer의 금융 분석 접목 29p

I. 서론

현재 AI 변화를 주도하는 대부분의 LLM 모델의 뿌리에는, "Attention Is All You Need" 논문의 Transformer 모델이 존재한다. AI 기술과 금융시장 분석과의 접목을 위해서, 지난 편에 이어 Attention 및 Transformer 모델에 대한 원리 설명을 완료하고자 한다. 본 자료에서는 Transformer 아키텍처의 실제 세부 구조에 대해서 하나씩 설명한다.

Transformer evolutionary tree



자료: https://blog.naver.com/jinp7

여기서 사용 및 설명하는 코드의 프로그래밍 환경은 다음과 같다.

```
Python 3.12.7
tensorflow 2.20.0
keras 3.12.0
```

II. Transformer 아키텍처

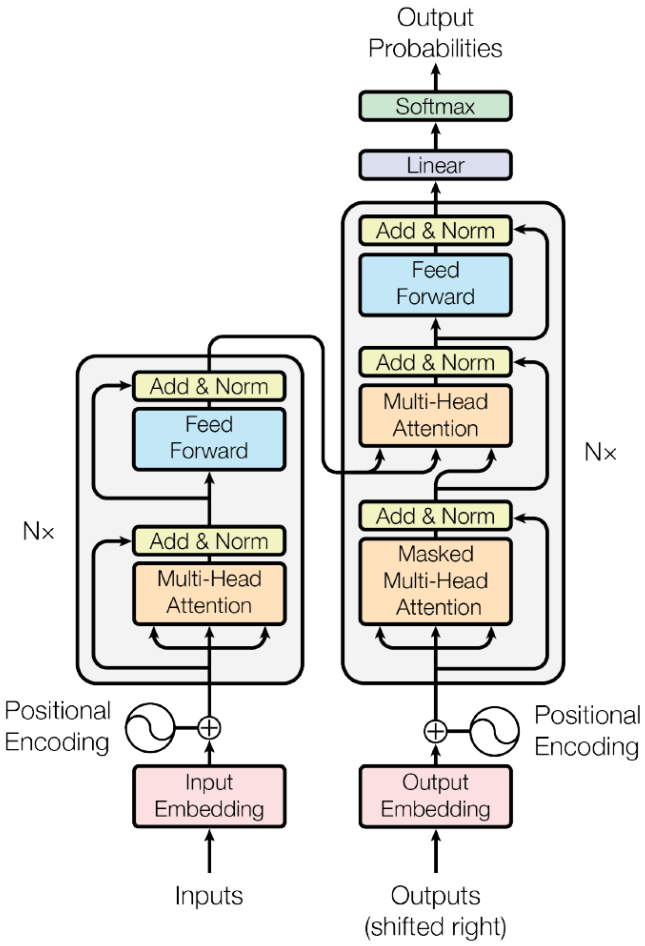
역사적인 논문 "Attention Is All You Need"에서의 Transformer 아키텍처는, 발표 당시 기계 번역 분야에서 SOTA(state of the art)를 달성했던 모델이다. 이후, 이 Transformer의 변형 모델이, 여러 산업의 다양한 응용 분야에서 활용되고 있다.

Transformer 아키텍처는, 발표 당시 오롯이 새로 개발한 여러 혁신적 알고리즘과 기존 기술들을 다양하게 접목함으로써 뛰어난 성능을 낼 수 있었다. 우선, 순환 신경망이 아닌 Attention 알고리즘만으로 구성된 인코더-디코더 형태를 제시한 것이 Transformer 모델의 혁신적인 변화라 할 수 있다. 또한 Multi-Head Attention, 'Sinusoidal' Positional Encoding 등도 이 논문에서 최초 개발한 알고리즘에 해당한다.

여기서는, Transformer 아키텍처의 전체 내용을 이해하기 위해, 이 안에서 사용된 다양한 알고리즘 및 기법 등에 대해 설명하고자 한다. 설명하는 알고리즘의 순서는, 원논문에서의 서술 순서를 동일하게 적용했다.

우선 Transformer 아키텍처의 전체 구조는 아래 그래픽과 같다.

Transformer 아키텍처의 전체 구조도

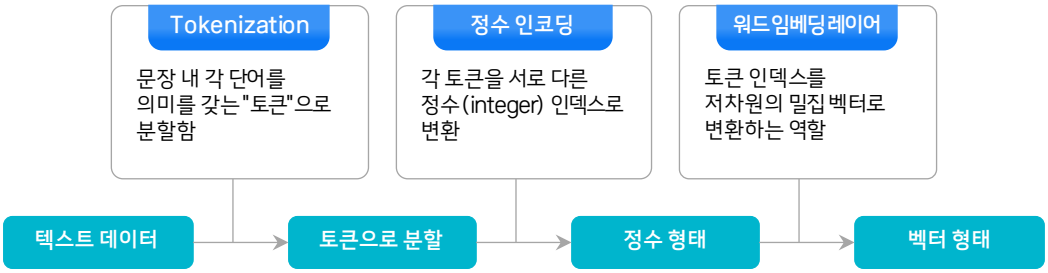


참고: 논문에서는 레이어 반복 횟수 N=6을 적용함
자료: "Attention Is All You Need" 논문

0. 임베딩 벡터 (자연어 처리 기초)

Transformer 아키텍처는 기계 번역의 목적으로 개발된 모델이다. 따라서, 해당 모델을 100% 이해하기 위해서는 자연어 처리(Natural Language Processing, NLP)에 대한 기본적인 이해가 필요하다. 여기서는 (지금껏 설명한 적이 없는) 자연어 처리의 필수 지식에 대해 간단히 정리한다.

기계가 텍스트 문장을 처리하기 위해서는 우선 다음과 변환 과정을 거친다.



위 단계 중 tokenization과 정수 인코딩 단계를 지나면 "한 줄의 텍스트 문장"이, 기계가 처리할 수 있는 최소한의 형태인 "토큰들의 정수 인덱스 데이터"로 바뀌게 된다. 이 두 단계는, 여러 가지의 기존 라이브러리를 통해서 손쉽게 처리할 수 있다. 예를 들어 keras.layers.TextVectorization 클래스를 사용하면, "the movie was a masterpiece"란 텍스트를 [2, 4, 3, 8, 5] 같은 숫자 리스트의 형태로 바꿔준다. 이 리스트는 "the" 토큰(단어)의 정수 인덱스가 2, "movie"의 정수 인덱스가 4로 기억되도록 각각 변환한 것이다.

그런데, 이 정수 인덱스 형태의 데이터를 그대로 신경망 모델에서 사용하는 것은 사실상 불가능하다. 이 정보를 만약 그대로 사용하게 되면, the의 인덱스 = 2의 정보와 movie의 인덱스 = 4의 정보로 인해서, "movie 단어는 the 단어의 곱하기 2와 같다"라는 의미 혹은 ordinal 관계가 모델 안에서 사용되는 "의미 왜곡"이 발생한다. 따라서, 각각의 토큰(단어)이 서로 다른 의미만을 가지는 형태가 되도록, 한 번 더 데이터 변환 과정을 거쳐야 한다.

원-핫 인코딩

이때 사용할 수 있는 하나의 선택지로 "원-핫 인코딩"이란 기법이 있다. "원-핫 인코딩"은 범주형 데이터를 수치형 데이터로 변환하는 기법으로, 각 범주를 이진 벡터 상의 고유한 위치에 1로 표시하고 나머지는 0으로 처리하여 범주 간의 순서 또는 크기 의미를 제거하도록 만들어주는 기법이다. 이는 계량경제학 분야로 비유하면, dummy 변수를 쓰는 방식과 사실상 같은 방식이다.

예를 들어, the, movie, was란 3개의 단어만 있다고 하면, the → [1, 0, 0], movie → [0, 1, 0], was → [0, 0, 1]의 각기 다른 3차원 벡터로 변환하는 방식이다. 그런데, 원-핫 인코딩에서는 단어의 종류만큼 one-hot 벡터의 크기를 키워야 한다. 즉, 단어 종류가 10,000개라면, one-hot 벡터의 차원은 10,000 차원으로 같이 커진다.

따라서, 이런 희소 벡터 형태의 표현법은 여러 가지 단점을 가지게 된다. 우선 토큰 집합 크기만큼 벡터 차원이 커지므로, 메모리 사용량과 계산 비용이 급증한다. 더 중요한 것은, 단어 간 의미 관계를 표현하지 못한다는 점이다. 여기서는 모든 단어가 완전히 독립적인 벡터로 표현되므로, 단어 간의 유사성, 반대 관계 등의 의미를 가지지 못한다. 이는 바로 뒤에 나올 워드 임베딩의 장점과 크게 대비되는 부분이다.

임베딩 레이어 / 워드 임베딩

최근의 자연어 처리 LLM 및 신경망 모델은, 텍스트 데이터를 신경망 안에서 "임베딩 레이어"로 변환해 사용하는 것이 일반적이다. (embed는 끼워넣다, 내장하다의 의미임.)

임베딩 레이어는, 텍스트 토큰을 "저차원 dense vector"로 표현하고 의미 관계를 벡터 공간에 반영하는 방식이다. 쉽게 말하면, 아주 많은 단어들을, 훨씬 적은 차원의 벡터 공간에서 서로 다른 벡터(한 점)로 매칭하는 형태다.

예로, "Attention Is All You Need" 논문에서 토큰(단어) 집합의 크기는 37,000개다. 그리고 임베딩 레이어의 벡터 차원은 512다. 따라서, 이는 37,000개의 단어를 $(1.2, 0.7, -0.3, \dots, 0.1), (-0.5, 1.3, -0.1, \dots, 0.3)$ 와 같은 "서로 다른 512차원 벡터 값"으로 표현하는 방식에 해당한다.

(반면에, 원-핫 인코딩에서는 단어 표현에 한 곳만 1, 나머지는 0으로만 구성된 37,000차원의 벡터가 동원됨.)

임베딩 레이어를 쓰면, 단어의 의미와 관계 정보를 벡터가 저장할 수 있게 된다. 쉽게 말해, Word2Vec 모델처럼 "아주 잘 학습된 임베딩 레이어"라고 하면, 각 단어 벡터를 가지고 [수식: king − man + woman]를 연산하면, "queen" 단어 벡터가 계산 결과로 나오는 방식의 연산이 가능하다.

이런 임베딩 레이어의 실제 내부 구조는, 사실 큰 사이즈의 정수 데이터가 들어오면 그에 맞는 저차원 벡터를 반환하는 룩업 테이블(Lookup Table) 형태의 가중치 행렬의 모습을 가진다. 즉, 이 레이어는 (n_voca, embed_dim) 모양의 큰 행렬 하나로 구성된다. 레이어는, 입력으로 단어의 정수 인덱스가 들어오면, 그 위치의 행 수치(저차원 벡터가 적힌)를 통째로 꺼내 오는 작업을 한다. 신경망 모델 내 임베딩 레이어의 초기화 시에는, 이 전체 Lookup 행렬이 랜덤한 수치로 채워진다. 각 신경망 모델이 정상적으로 학습(역전파) 과정을 진행하게 되면, loss를 줄이기 위해서 임베딩 레이어 행렬 수치도 계속 업데이트가 된다. 이 과정이 진행되면서, 학습 데이터가 가지고 있는 도메인의 특성에 맞춰서 단어의 뉘앙스가 벡터에 계속 투영이 된다.

만약, 신경망 모델이 "영화 리뷰 텍스트의 긍정, 부정 분류" 데이터를 사용하면, 학습이 진행되면서 유사한 '긍정적 감성'의 단어들은 단어 벡터가 서로 가까운 위치로 모이게 된다. 반대로 '부정적 감성'의 단어들은 긍정 단어 벡터와 멀리 떨어진 위치로 점차 이동하게 된다.

만약 신경망 모델이 매우 정교한 기계 번역 등을 수행한다고 하면, 학습이 진행되면서 실제 단어 뜻의 유사도 등에 따라서, 단어 벡터의 위치가 정교하게 이동하게 된다. 이런 학습이 진행되면, 단어 벡터로 [king − man + woman = queen]과 같은 형태의 연산이 가능해진다.

이런 작동법을 예시 코드를 통해 알아보자.

임베딩 레이어 학습 원리 예시 코드

```
import random
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, datasets, preprocessing

SEED = 42
random.seed(SEED)
tf.random.set_seed(SEED)
tf.config.experimental.enable_op_determinism()

def get_cosine_similarity(v1, v2):
    return np.dot(v1, v2) / (np.linalg.norm(v1) * np.linalg.norm(v2))

# 1. 데이터 로드 및 전처리
max_features = 10000  # 사용할 단어 사전의 크기
maxlen = 200          # 리뷰 한 개당 단어 수 (길이 맞추기)

(x_train, y_train), (x_test, y_test) = datasets.imdb.load_data(num_words=max_features)
word_index = datasets.imdb.get_word_index() # IMDB data 특성 상 word.index.get 사용 시 +3 offset 필요함
x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)

# 2. 모델 구성
model = tf.keras.Sequential([
    # Embedding 층: 10,000개의 단어를 16차원의 밀집 벡터로 매핑
    layers.Embedding(input_dim=max_features, output_dim=16),
    layers.GlobalAveragePooling1D(), # 벡터의 평균을 취해 차원 축소
    layers.Dense(16, activation='relu'),
    layers.Dense(1, activation='sigmoid') # 0(부정) 또는 1(긍정) 출력
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.build(input_shape=(None, maxlen))
model.summary()

# 3. 학습 전 워드 임베딩 벡터 확인 (무작위 상태)
print('\n(BEFORE)')
weights_before = model.layers[0].get_weights()[0]
vec_good_before = weights_before[word_index.get('good')+3]
vec_bad_before = weights_before[word_index.get('bad')+3]
print('good:', vec_good_before[:7])
print('bad:', vec_bad_before[:7])
print('Cosine Similarity:', get_cosine_similarity(vec_good_before, vec_bad_before), '\n')

# 4. 모델 학습
model.fit(x_train, y_train, epochs=5, batch_size=32, validation_split=0.2)

# 5. 학습 후 워드 임베딩 벡터 확인 (변화된 상태)
print('\n(AFTER)')
weights_after = model.layers[0].get_weights()[0]
vec_good_after = weights_after[word_index.get('good')+3]
vec_bad_after = weights_after[word_index.get('bad')+3]
print('good:', vec_good_after[:7])
print('bad:', vec_bad_after[:7])
print('Cosine Similarity:', get_cosine_similarity(vec_good_after, vec_bad_after))
```

위 코드는, 유명한 "IMDB 사이트의 영화 리뷰 글과, 그 결론을 긍정 혹은 부정으로 분류한 데이터"를 가지고 학습하여, 신규 텍스트에 대해 긍정 혹은 부정의 분류를 할 수 있는 신경망 모델을 만드는 간단한 예시다.

이때, 신경망의 구조는 간단히 "임베딩 → GlobalAveragePooling1D → Dense → Dense"의 형태로 되어 있다 (GlobalAveragePooling1D 레이어는 26/1/8일 자 "CNN AI 투자 모델" 리포트를 참고).

처음에 입력 데이터를 지정할 때는 tensorflow.keras.datasets.imdb.load_data를 사용하는데, 이는 IMDB 리뷰 텍스트 문장들을 '토큰의 정수 인덱스 변환'까지 끝난 상태로 제공하는 코드다.

위 전체 코드를 실행하면 먼저, 신경망 모델의 학습 이전 단계에서 "good"과 "bad" 단어의 임베딩 벡터 앞부분을 (7번째 자리까지만) 한 번 프린트한다. 그리고, 학습 데이터로 일정 규모의 학습을 진행하고, 그다음에 "good"과 "bad" 단어의 임베딩 벡터 앞부분을 또 한 번 프린트한다. 매번 두 벡터 간의 코사인 유사도도 계산 및 출력한다.

이 전체 코드의 실행 결과는 다음과 같다.

```
>

(BEFORE)
good: [-0.01587968  0.014062    0.01974915  0.01822606 -0.03160763 -0.00028857  -0.01255069]
bad: [-0.00632775 -0.01728451 -0.01178502 -0.02647623  0.0115847   0.02343085   0.02774663]
Cosine Similarity: -0.17741615

Epoch 1/5
625/625 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6s 7ms/step - accuracy: 0.7460 - loss: 0.5458 - val_accuracy:
0.8564 - val_loss: 0.3729
Epoch 2/5
625/625 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - accuracy: 0.8786 - loss: 0.3044 - val_accuracy:
0.8776 - val_loss: 0.3039
Epoch 3/5
625/625 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - accuracy: 0.9079 - loss: 0.2369 - val_accuracy:
0.8802 - val_loss: 0.2913
Epoch 4/5
625/625 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - accuracy: 0.9251 - loss: 0.1985 - val_accuracy:
0.8816 - val_loss: 0.2937
Epoch 5/5
625/625 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6s 7ms/step - accuracy: 0.9388 - loss: 0.1706 - val_accuracy:
0.8782 - val_loss: 0.3052

(AFTER)
good: [ 0.26454827 -0.09980083  0.32498604 -0.24328858 -0.3075287   0.2686854  -0.30797   ]
bad: [-0.6296084   0.5813321  -0.5761308   0.5505309   0.6635738  -0.59093523  0.6197061 ]
Cosine Similarity: -0.90744656

>
```

신경망 설정 당시, 단어 임베딩 벡터(임베딩 레이어의 가중치 행렬 수치)는 단순 랜덤 수치로 채워져 있다. 따라서, good 단어와 bad 단어의 임베딩 벡터 수치는 각 원소가 비슷한 경우도 있고 떨어진 경우도 있으며 특정 패턴이 존재하지 않는다.

신경망 모델을 model.fit으로 학습을 진행하게 되면, 각 문장들이 긍정, 부정으로 잘 분류되도록 신경망 내 모든 위치의 가중치 행렬이 업데이트된다. 이에 따라, 감성 분석(긍정/부정)의 목적에 맞도록 임베딩 레이어의 벡터 위치 또한 계속 수정이 진행된다. 이에 따라, 긍정 뉘앙스의 단어 벡터들은 서로 모이게 되고, 부정 뉘앙스의 단어 벡터들은 이와 떨어진 위치에서 서로 모이게 된다.

실제 학습 후의 결과를 보자. 학습 후에는, 성격이 전혀 다른 good 단어와 bad 단어의 위치는 매우 멀어지도록 바뀌어져 있다. 현재 good 벡터의 1~7번째 원소와 bad 벡터의 1~7번째 원소는 전체 다 부호가 서로 반대면서 수치 또한 커졌다. 벡터의 나머지 원소 값도 확인해야 봐야 하겠지만, 두 단어 벡터의 위치가 전보다 상당히 멀어졌다고 우선 예상할 수 있다. 실제로 두 벡터 간의 유사도를 측정하는 '코사인 유사도' 계산을 해보면, 학습 후에는 코사인 유사도가 -0.907로 감소하여, 두 벡터의 방향이 상당히 상반됨이 확인된다. 즉, 이는 감성 분석 측면에서 두 단어가 상반됨을, 임베딩 레이어가 학습한 결과라고 볼 수 있다. 이것이 바로 임베딩 레이어가 학습하는 방식이다.

좀 더 설명하면, 이 학습된 임베딩 공간에서 "good" 벡터에서 "bad" 벡터를 빼면, 이 벡터가 바로 "good – bad 감정 축(sentiment axis)"이 된다고 볼 수 있다.

$$\text{sentiment axis} = \text{embedding}(\text{“good”}) - \text{embedding}(\text{“bad”})$$

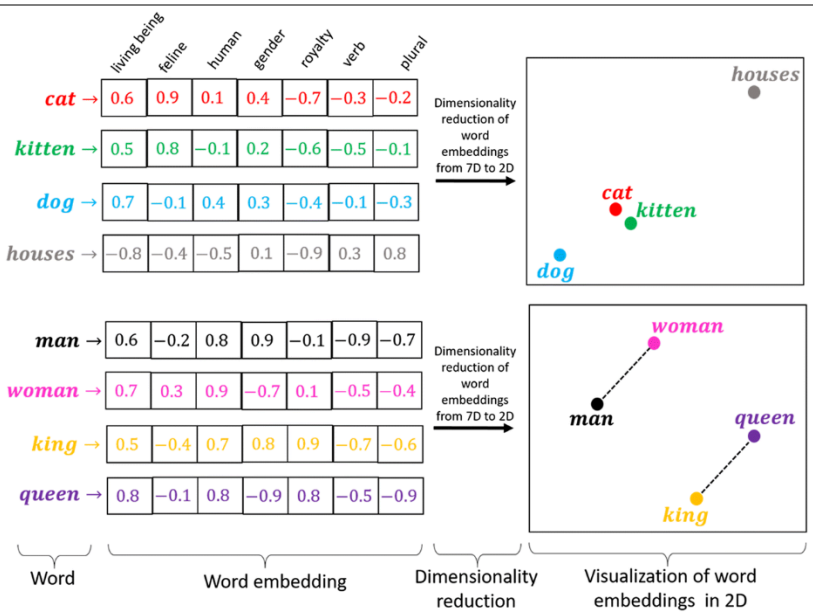
이 sentiment axis 벡터는, "긍정 방향"으로의 변화를 나타내는 축이 된다.

만약에 "나쁜 징조"란 의미의 한 단어가 있고, "좋은 징조"란 의미의 신조어가 하나 있다고 해보자. 이때, "나쁜 징조(omen)"의 단어 벡터에 이 sentiment axis 벡터를 더하면, "좋은 징조"의 단어 벡터와 거의 유사한 벡터가 도출되는 식으로 작동한다. (물론 이런 sentiment axis는 학습 데이터에 따라서 다르게 구성됨.)

이처럼 임베딩 레이어는, 의미상으로 유사한 단어들이 벡터 공간에서 서로 가까운 위치에 배치될 수 있도록 해주는 역할을 한다. 현재에서는 LLM 및 모든 형태의 텍스트 처리 신경망 모델에서 이 임베딩 레이어가 거의 필수적으로 포함된다.

Transformer 전체 구조도에서 이 임베딩 레이어는, 왼쪽 인코더 파트에는 "Input Embedding"으로, 오른쪽 디코더 파트에는 "Output Embedding"란 이름으로 맨 첫 단계에 적용되어 있다.

워드 임베딩의 개념



자료: https://velog.io/@leejaejun

1. Residual Connection, Layer Normalization (#3.1 / 기존 기술)

(괄호 안 #3.1은, 원논문에서 항목 번호 3.1 위치에 해당 내용이 들어 있음을 나타냄. 이후 동일하게 계속 적용.)

앞서 Transformer 전체 구조도를 크게 보면, 번역의 입력 문장을 처리하는 '왼쪽의 인코더 파트'와 출력 문장을 처리하는 '오른쪽의 디코더 파트'로 나뉜다. 인코더 파트는, Input Embedding과 Positional Encoding(뒤에서 설명)이 끝난 다음에, 메인 레이어를 6번을 반복한다(Nx로 표시된 부분). 이때 하나의 메인 레이어는, Multi-Head Attention 레이어(뒤에서 설명)와 Position-wise Feed Forward 레이어(뒤에서 설명)라고 하는 2개의 서브 레이어로 구성된다.

이때, 각 해당 서브 레이어들을 그대로 붙이는 방식이 아니라, 각 서브 레이어의 출력을 아래의 수식처럼 1) residual connection을 먼저 하고 2) 그다음에 레이어 정규화를 적용하는 방식을 사용한다.

$$LayerNorm(x + Sublayer(x))$$

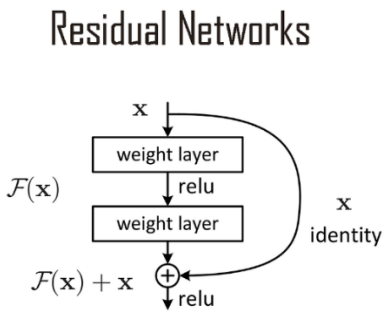
이 절차는, 구조도 그래픽에서 1) "Add & Norm"이란 용어와 2) 화살표가 두 갈래로 갈라졌다가 다시 합쳐지는 모양이 의미하는 것과 동일한 내용이다.

Residual Connection [$x + Sublayer(x)$]

Residual Connection 혹은 Residual Network(ResNet)는 2015년 Kaiming He 등이 제안한 아키텍처로, 신경망을 단순히 층층이 쌓는 것이 아니라, 받은 입력 데이터를 뒤쪽 층에 직접 전달하는 지름길(skip connection)을 같이 만드는 방식을 말한다.

ResNet은 $H(x) = F(x) + x$의 신경망 구조를 가진다. 이 수식을 $F(x) = H(x) - x$의 형태로 보면, 이는 잔차(residual)에 대해서 학습하는 신경망으로 볼 수 있기 때문에, 이 신경망이 Residual이란 명칭을 가지게 되었다.

Residual Connection 구조



자료: https://medium.com/@AnasBrital98

ResNet은 잔차 학습의 정교성을 가지고 있기 때문에, 신경망 층을 매우 깊게 쌓으면서도 모델을 정교하게 학습시켜야 할 때 널리 사용되는 알고리즘 중의 하나다. 실제로 이 ResNet 구조로 인해서 전체 sublayer를 여러 번 깊게 쌓는 것이 가능하다고 할 수 있다. 따라서, ResNet 구조 또한 Transformer에서 상당히 중요한 역할을 한다고 할 수 있다.

레이어 정규화 [$LayerNorm(\,)$]

레이어 정규화(층 정규화, Layer Normalization)는 각 입력 샘플의 내부에서, 한 샘플의 모든 feature를 대상으로 하여 평균과 분산을 정규화하는(평균 0, 분산 1로) 방법을 말한다. 이 정규화는 각 샘플(그리고 각 timestep)마다 평균, 분산을 따로 모두 계산한다. 안정적인 신경망 학습을 위해서 데이터의 정규화는 항상 필요한데, Transformer에서는 문장 길이(시퀀스 길이)가 다양할 수 있으므로 '배치 정규화' 방법이 어려운 측면이 있다. 이런 이유 등으로 인해서 모델에서는 그 대신 '레이어 정규화'를 사용한다.

Transformer 당시의 레이어 정규화 방식은 Post-LN(Layer Normalization): $LN(x + Sublayer(x))$ 방식이었다. 그러나, 최근에는 그래디언트가 훨씬 안정적으로 흐르는 Pre-LN: $x + Sublayer(LN(x))$ 방식이 많이 사용되고 있다.

참고로, 배치 정규화(batch norm)란 기법은, 각 미니배치 데이터에서 feature별로 정규화하는 방식을 말한다. 또한 Feature Scaling 기법(StandardScaler과 같은)이라고 하여, 전체 데이터에 대해서 feature별로 정규화를 진행하는 전처리 기법 또한 따로 존재한다. 각 정규화 기법의 차이를 정리하면 다음과 같다.

정규화 기법들의 비교

기법	평균 계산 기준
Feature Scaling(StandardScaler 같은 전처리)	전체 데이터의 같은 feature 대상
Batch Norm	미니배치의 같은 feature 대상
Layer Norm	한 샘플의 모든 feature 대상

자료: 삼성증권

2. Scaled Dot-Product Attention (#3.2.1 / 신규 기술)

Transformer 모델에서는 Scaled Dot-Product Attention 알고리즘을 개발해서 사용하는데, 이의 수식은 다음과 같다.

우선, Transformer에서의 Attention 레이어는, Attention 알고리즘을 사용해 모든 토큰에 대해서 "단어 연관성을 감안한(가중치와 단어 벡터를 가중합한) Value 기준의 문맥 반영 벡터"를 리턴하는 레이어라고 할 수 있다.

$$Attention(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = softmax\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{\boldsymbol{T}}}{\sqrt{d_k}}\right)\boldsymbol{V}$$

이때, $\boldsymbol{Q}$는 행렬로 표시한 query 벡터의 모든 집합, $\boldsymbol{K}$는 행렬로 표시한 key의 모든 집합, $\boldsymbol{V}$는 행렬로 표시한 value의 모든 집합, d_k는 key 벡터의 차원을 뜻함.

여기서의 Scaled Dot-Product Attention 방식은, 유사도 계산법의 관점에서 본다면 Luong의 dot-product Attention과 계산법이 동일하다. (2월 19일 자 "퀀트 모델링 A to Z - (10) Attention/Transformer의 이해, 1부" 리포트 참고).

Luong Attention 대비 Scaled Dot-Product Attention에서의 바뀐 점은 1) Attention Score 계산 시 $\sqrt{d_k}$로 나눈다는 점, 2) RNN 방식을 버림으로써, 모든 토큰의 Attention Output 계산을 한 번의 행렬 계산으로 끝낸다는 점이다. 1번 항목의 변화는 크지 않지만, 2번 항목의 변화는 대규모 병렬 처리를 가능하게 하여 모델 학습 속도를 획기적으로 올렸다는 것이 매우 큰 발전이다. 실제로 Transformer 모델의 가장 큰 의의는, 처음으로 RNN 방식을 버리고 병렬 기반의 Attention 알고리즘만으로 모든 처리를 진행한다는 점이라고 볼 수 있다.

한편, 논문의 서술상에는 Multi-Head Attention 파트에 기록되어 있지만, Transformer에 사용된 모든 Attention 알고리즘에서는 항상 query, key, value 벡터에 각각 학습 가능한 가중치 행렬을 곱해서 다양한 문맥 정보를 탐색할 수 있도록 했다. 이 또한 Luong dot-product Attention과 획기적으로 달라진 부분이다. (이에 대해서는 뒤의 Multi-Head Attention 목차에서 설명함.)

Attention 관련 용어

논문의 Attention 관련 용어는, 기존과 약간 다르게 표기되고 있다. 이를 초기 Attention에서의 용어와 비교하면 다음과 같다.

Attention 알고리즘 관련 용어

의미	초기 Attention의 용어	Scaled Dot-Product Attention의 용어
단어 간의 단순 연관성 점수	Attention Score	$\frac{\boldsymbol{Q}\boldsymbol{K}^{\boldsymbol{T}}}{\sqrt{d_k}}$
연관성 점수를 0~1의 가중치로 변환한 수치	Attention Weight	$softmax\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{\boldsymbol{T}}}{\sqrt{d_k}}\right)$
단어 연관성을 감안한(가중치와 단어 벡터를 가중합한) 문맥 반영 벡터	Context Vector	Attention Output = $softmax\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{\boldsymbol{T}}}{\sqrt{d_k}}\right)\boldsymbol{V}$

참고: 2월 19일 자 "퀀트 모델링 A to Z - (10) Attention/Transformer의 이해, 1부" 리포트를 참고

기본적인 계산 절차는, 1) 단어 간 연관성 점수를 계산하고, 2) 이를 0~1의 가중치로 변환하고, 3) 단어 벡터와 이에 해당하는 가중치를 가중 합(가중 평균)한 문맥 반영 벡터를 만드는 순서이다.

논문에서의 $\boldsymbol{Attention(Q,K,V)}$ 용어는, 이 절차의 마지막 문맥 반영 벡터를 의미한다. 논문은 연관성 점수, 가중치 수치, 문맥 반영 벡터 단계를 전부 수식으로 압축하여, $\boldsymbol{Attention(Q,K,V)}$의 수식 하나만을 남긴 것이 특징이다.

Query, Key, Value

이 논문의 Attention 수식을 보면, 입력 변수에 Query, Key, Value라는 용어가 나온다. Query란 용어는 데이터베이스/인터넷 검색에서의 검색어에 비유한 용어다. 즉, 연관성 정보를 찾으려고 하는 디코더의 현재 단어에 해당하는 변수다(논문은 실제로는, 행렬로 처리하기 때문에 디코더의 모든 단어 행렬을 사용함). Key, Value란 용어는 Python의 딕셔너리 자료형의 용법과 비슷하다. Key가 실제 찾으려는 자료의 표제, Value가 실제 자료를 나타내며, Key로 조회하면 Value를 리턴하는 방식이다. 이런 구분된 변수는 비교적 초기 Attention에도 모두 존재했던 개념이라고 할 수 있다. 각 용어를 좀 더 정리하면 다음과 같다.

Query, Key, Value의 비유적 의미

용어	비유적 표현	심플한 Cross-Attention 상의 데이터
Query	내가 찾고자 하는 검색어	디코더 상의 한 단어 벡터 (혹은 단어 전부)
Key	책의 제목	인코더 상의 모든 단어 벡터
Value	책의 내용	인코더 상의 모든 단어 벡터

참고: 논문의 실제 Attention에서는, Query에 한 단어 벡터가 아닌, 전체 단어 행렬을 사용함. 또한 Q, K, V에 각각 가중치 행렬을 곱하여 사용함.

만약 Microsoft Excel 프로그램의 사용자라면, index/match 함수 혹은 vlookup 함수의 사용 시에 Query, Key, Value 구조의 개념을 쉽게 알 수 있다.

Excel의 Index/match 함수 예시

```
=INDEX(C:C,MATCH("A005930",A:A,0))
← A005930을 A열에서 찾은 다음, 그 위치를 C열에 대입해서 해당 위치의 값을 리턴하라.
=INDEX(Value,MATCH(Query,Key,0))
```

위 Excel 예시의 Index/match 함수 조합에 들어가는 입력 값 들은, 구조로만 본다면 Query, Key, Value와 완벽히 일치한다.

Excel의 vlookup 함수 예시

```
=VLOOKUP("A005930",A:C,3)
← A005930을 'A:C 범위의 첫 열'(A열)에서 찾은 다음, 그 위치를 'A:C 범위의 3번째 열'(C열)에 대입해서 해당 위치의 값을 리턴하라.

=VLOOKUP(look_val,table,col_idx_num)
← look_val이 Query, table의 첫 열이 Key, 'table의 col_idx_num번째 열'이 Value에 해당
```

예를 들어, "I'm not sad."의 영어 문장과 "Je ne suis pas triste."의 프랑스어 문장의 단어 간 연관성을 계산한다고 하자. 이번 계산에서 "sad" 단어와 프랑스어 문장 각 단어와의 Attention value들을 계산한다고 하면, 이때 이번 query에 해당하는 것이 "sad" 단어다. 이때 Key는 "Je ne suis pas triste."의 각 단어 벡터의 집합(행렬)이 된다. Value 또한 "Je ne suis pas triste."의 각 단어 벡터의 집합(행렬)이다.

참고로, 직전 리포트에서 나왔던 Bahdanau Attention에서는 query가 $\boldsymbol{s_t}$를 말한다. 이때, key와 value는 $\boldsymbol{H}$다. [$score(\boldsymbol{s_t}, \boldsymbol{H}) = \boldsymbol{v_a}^T \tanh(\boldsymbol{W_1 s_t} + \boldsymbol{W_2 H})$]. Luong Attention에서는 query가 $\boldsymbol{h_t}$이고 key와 value는 $\bar{\boldsymbol{h}}_{\boldsymbol{s}}$를 말한다. [$score(\boldsymbol{h_t}, \bar{\boldsymbol{h}}_{\boldsymbol{s}}) = \boldsymbol{h_t}^T \bar{\boldsymbol{h}}_{\boldsymbol{s}}$].

Attention 수식 설명

$$Attention(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = softmax\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}}\right)\boldsymbol{V}$$

Scaled Dot-Product Attention은 말 그대로 dot-product 방식의 Attention 계산법을 쓰므로, 전에 보았던 Luong Attention과 동일한 내적 방식이다. 단, Transformer 모델은 RNN을 없앤 방식이므로, Query에 데이터를 넣을 때, 한 timestep씩 한 토큰을 넣는 것이 아니고, 문장의 모든 토큰을 동시에 입력시킬 수 있다. 즉, Query에도 행렬이 입력되는 방식이다. 이 행렬 계산을 축약해서 표기한 것이 위의 Attention 수식이다. 이를 풀어서 쓰면 다음과 같다.

[변수 가정 및 설명 (한 개 샘플 기준의 설명)]

d: Query, Key, Value의 각 토큰을 기록한 벡터의 차원(Multi-Head Attention이 아니라고 가정하면, 가중치 행렬 적용 전의 embedding_dim을 의미함). d_k도 d를 의미함

n: Query에 해당하는 (보통 디코더의) 한 문장의 토큰 수

$\boldsymbol{Q}$: Query에 해당하는 (보통 디코더의) 한 문장의 모든 토큰 정보. (n, d) 형태의 행렬임 ($\boldsymbol{Q} \in \mathbb{R}^{n \times d}$)
쉽게 말해 d차원의 행 벡터로 표현되는 단어들을 n개를 밑으로 쌓아서, 행렬 형태로 만든 것임.

$$\boldsymbol{Q} = \begin{bmatrix} q_{11} & \cdots & q_{1d} \\ \vdots & \ddots & \vdots \\ q_{n1} & \cdots & q_{nd} \end{bmatrix}$$

m: Key, Value에 해당하는 (보통 인코더의) 한 문장의 토큰 수

$\boldsymbol{K}$: Key에 해당하는 (보통 인코더의) 한 문장의 모든 토큰 정보. $\boldsymbol{K} \in \mathbb{R}^{m \times d}$

$\boldsymbol{V}$: Key와 동일한 (보통 인코더의) 한 문장의 모든 토큰 정보. $\boldsymbol{V} \in \mathbb{R}^{m \times d}$

$$\boldsymbol{K} = \begin{bmatrix} k_{11} & \cdots & k_{1d} \\ \vdots & \ddots & \vdots \\ k_{m1} & \cdots & k_{md} \end{bmatrix}, \boldsymbol{V} = \begin{bmatrix} v_{11} & \cdots & v_{1d} \\ \vdots & \ddots & \vdots \\ v_{m1} & \cdots & v_{md} \end{bmatrix}$$

이 경우, 전체 수식을 하나씩 분해해서 점검하면 다음과 같다.

QK^T:

$$\boldsymbol{QK^T} = \begin{bmatrix} q_{11} & \cdots & q_{1d} \\ \vdots & \ddots & \vdots \\ q_{n1} & \cdots & q_{nd} \end{bmatrix} \begin{bmatrix} k_{11} & \cdots & k_{m1} \\ \vdots & \ddots & \vdots \\ k_{1d} & \cdots & k_{md} \end{bmatrix}$$

$$= \begin{bmatrix} (q_{11}k_{11} + q_{12}k_{12} + \cdots + q_{1d}k_{1d}) & \cdots & (q_{11}k_{m1} + \cdots + q_{1d}k_{md}) \\ \vdots & \ddots & \vdots \\ (q_{n1}k_{11} + \cdots + q_{nd}k_{1d}) & \cdots & (q_{n1}k_{m1} + \cdots + q_{nd}k_{md}) \end{bmatrix}$$

수식을 보면 $\boldsymbol{QK^T}$의 각 원소는, Query의 모든 행 벡터와 Key의 모든 행 벡터(혹은 Key transpose의 모든 열 벡터)의 내적 결과를 하나씩 담고 있다. 즉, 이 $\boldsymbol{QK^T}$는 이 내적 결과들을 하나의 행렬로 표기하는 방법에 해당한다.

다시 말해, $\boldsymbol{QK^T}$는 Query의 모든 n개 토큰과 Key의 모든 m개 토큰 간의 내적값을 한 번에 (n×m)의 행렬로 보여주는 역할을 한다. Luong 내적 계산의 행렬 버전이라고 볼 수 있다.

$\frac{QK^T}{\sqrt{d_k}}$:

이 논문의 Scaled Dot-Product Attention에서는, $\boldsymbol{QK^T}$수치를 그대로 쓰지 않고 $\boldsymbol{QK^T}$를 $\sqrt{d_k}$로 나눠 준다. 이것은 softmax gradient 안정화(saturation 방지) 혹은 $\boldsymbol{QK^T}$ 원소 값의 표준화를 위해 추가한 것이다. 논문에 따르면 q, k의 각 성분이 평균 0, 분산 1이고 서로 독립인 확률 분포라면 q, k의 dot-product는 평균 0, 분산 d_k이 된다. 즉, $1/\sqrt{d_k}$ 연산은 softmax gradient 안정화 혹은 표준화를 위한 조정에 해당한다.

이는 필자가 과거에 썼던 "매크로 다이내믹 모델"[1]에서의 표준화용 계수 조정과 비슷한 기법에 해당한다. 2013/4/23일 자 "매크로 기반 Dynamic Quant 모델" 리포트를 보면, 두 시점 간 유사도 계산을 위한 Statistical Distance 변수 생성에서 6차원(6개월 정보) 벡터를 사용하면서 표준화를 위해 $1/\sqrt{6}$의 조정법을 쓴 바이다. 이 둘의 방식은 같은 원리라 할 수 있다.

$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)$:

softmax 함수는, 잘 알다시피 K개의 실수 튜플을 입력으로 받아서, K개의 확률로 구성된 확률 분포로 정규화하는 함수다. 함수식은 $[softmax(x_i) = e^{x_i}/(\sum_i e^{x_i})]$다.

softmax를 적용하면, 내적 기반의 초기 수치에서 디코더의 토큰별로 가중치 합계가 1이 되는 Attention Weight 정보로 바뀐다. 이는, $\mathbb{R}^{n\times m}$의 행렬이다.

$$softmax\left(\frac{\boldsymbol{QK^T}}{\sqrt{d_k}}\right) = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix}$$

이때, a_{ij}는 Query의 i번째 토큰에 대한 Key의 j번째 토큰의 연관도 가중치(Attention Weight).

행렬의 모든 행 기준의 원소 합계는 1임(Query의 i번째 토큰에 대한 Attention Weight들의 합계는 항상 1).

1 한편, 2013년 당시의 "매크로 다이내믹 모델"을 현재의 머신 러닝 관점에서 보자면, 해당 모델은 러프한 KNN(K-Nearest Neighbors) 머신 러닝 기법을 활용한 투자 모델에 해당한다.

$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$:

마지막 단계는 단어 연관도(Attention Weight)와 Value의 단어들 벡터를 사용해서, Query의 각 단어에 대한 Value 기준의 문맥 반영 벡터를 만드는 단계다(논문 기준에서는 Attention output).

$$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix}\begin{bmatrix} v_{11} & \cdots & v_{1d} \\ \vdots & \ddots & \vdots \\ v_{m1} & \cdots & v_{md} \end{bmatrix}$$

$$= \begin{bmatrix} (a_{11}v_{11} + a_{12}v_{21} + \cdots + a_{1m}v_{m1}) & \cdots & (a_{11}v_{1d} + \cdots + a_{1m}v_{md}) \\ \vdots & \ddots & \vdots \\ (a_{n1}v_{11} + \cdots + a_{nm}v_{m1}) & \cdots & (a_{n1}v_{1d} + \cdots + a_{nm}v_{md}) \end{bmatrix}$$

이때, a_{ij}는 Query의 i번째 토큰에 대한 Key의 j번째 토큰의 연관도 가중치(Attention Weight)임.

$\boldsymbol{V}$는 Key와 동일한 (보통 인코더의) 한 문장의 모든 토큰 정보임. $\boldsymbol{V} \in \mathbb{R}^{m \times d}$

이 최종 결과는 $\mathbb{R}^{n \times d}$ 행렬로, Query의 각 토큰에 대한 Value 기준의 합성된 문맥 반영 벡터를 가지는 행렬이다. 행렬의 각 행이, 한 개 토큰 기준의 문맥 반영 벡터를 뜻한다.

예를 들어 보자. Query 2번 토큰(단어)에 대한 새로운 문맥 벡터의 첫 번째 성분(z_{21})을 구하면 다음과 같다.

$$z_{21} = a_{21}v_{11} + a_{22}v_{21} + \cdots + a_{2m}v_{m1}$$

a_{21}: Query 2번 토큰이 Value 1번 토큰에 집중하는 비중, a_{22}: Query 2번 토큰의 Value 2번 토큰 집중 비중

v_{21}: Value 1번 토큰의 첫 번째 성분(1번 토큰의 임베딩 벡터에서 첫째 위치), v_{22}: Value 2번 토큰의 첫째 성분

새로운 문맥 벡터의 세 번째 성분(z_{23})을 구하면 다음과 같다.

$$z_{23} = a_{21}v_{13} + a_{22}v_{23} + \cdots + a_{2m}v_{m3}$$

이런 식으로 z_{2j}의 수치들이 모두 결정되면서, Query 2번 토큰(단어)에 대한 새로운 문맥 벡터가 완성된다. 기계 번역 측면에서 보자면, 독일어 문장(Query) 2번째 단어가 가장 집중해야 하는(번역의 대상이 되는) 영어 문장(Value)의 연관 단어들의 가중합으로 만든 문맥 벡터를 리턴하는 형태다. 이런 학습의 노하우를 습득 및 저장하는 방법은, 다음 Multi-Head Attention 파트에서 상세히 설명한다.

지금까지의 내용을 보면, $softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$란, 기존 Attention 알고리즘에서 Attention Score → Attention Weight → Context Vector까지 계산하던 것을 단 하나의 수식으로 표현한 것에 불과함을 알 수 있다(Query 토큰 전체에 대해 한 번에 계산하도록 행렬 형태를 취한 것이 특징).

즉, Scaled Dot-Product Attention 기술은, 행렬 계산으로 표현된 $softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$ 수식과 완전히 동일한 내용을 뜻한다.

3. Multi-Head Attention (#3.2.2 / 신규 기술)

우선, 지금까지 배운 딥러닝 모델 학습 내용을 통해서, "가중치 행렬"이란, 모델이 수많은 데이터를 겪으면서 터득한 '판단 기준' 혹은 '노하우'가 수치로 기록된 저장소에 해당함을 알 수 있다.

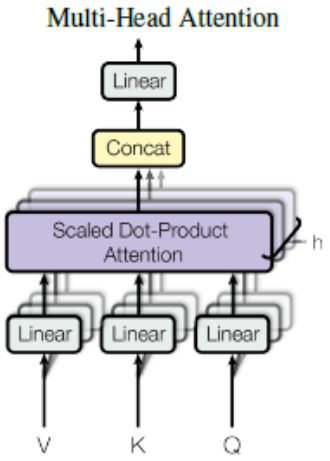
이전의 Luong 'dot-product' Attention(Luong 논문상의 대표적인 Attention 방식)에서는, Attention 정보의 계산법은 있지만 Attention 전용의 가중치 행렬은 없었다(Luong의 여러 variant 중 dot-product 방식에 한정했을 때만). 대신 Luong dot-product Attention을 가지고 있는 "전체 LSTM/GRU 모델" 상의 순환 가중치 행렬이, 해당 노하우들을 기록하는 역할을 했다. Luong Attention과 달리, Transformer 모델 구조에서는 LSTM/GRU 구조가 없기 때문에, Attention 작업과 관련한 노하우를 기록하는 별도의 가중치 행렬이 따로 필요하다고 할 수 있다.

이에 Transformer 모델은, Attention 작업 시 맨 첫 단계에서 Query, Key, Value 행렬 모두에 Linear Projection 가중치 행렬을 항상 곱하여 넣도록 했다. 이 가중치 행렬이 Attention 작업과 관련한 노하우를 저장하는 저장소가 된다.

즉, Transformer 모델의 Attention에서는 $\boldsymbol{Q}$ 대신 '가중치 행렬로 변환한' $\boldsymbol{QW^Q}$ 이 들어가고, $\boldsymbol{K}$ 대신 $\boldsymbol{KW^K}$ 이 들어가고, $\boldsymbol{V}$ 대신 $\boldsymbol{VW^V}$ 이 입력으로 들어간다. 모델 학습에 의해서 이 $\boldsymbol{W^Q}, \boldsymbol{W^K}, \boldsymbol{W^V}$ 란 가중치 행렬들이 Attention과 관련된 노하우를 기록하게 된다.

한편, 문장 내에서 각 단어 간의 관계/연관성을 조사할 때, 꼭 하나의 패턴으로만 찾을 필요는 없다. 마치 CNN 모델에서 CNN이 여러 개의 필터를 사용하여 여러 가지 패턴을 감지하듯이, Attention도 한 개 계산 구조(필터)가 아닌 여러 개의 계산 구조(필터)를 사용하면, 각 단어 간의 다양한 측면의 연관 관계를 학습할 수 있다. 따라서, Attention 레이어에서도 동일 계산 구조를 여러 개 만들어서 붙임으로써, 문맥 이해를 여러 차원에서 종합적으로 할 수 있게 만든 것이 바로 Multi-Head Attention이다. Multi-Head Attention 하의 각각의 개별 Head는, 예를 들어 하나는 단어들의 문법 구조에만 집중하는 역할을 가지고, 다른 하나는 의미적 유사성에 집중하는 역할을 가지는 등 서로 다른 연관성을 탐지하는 역할을 한다.

Multi-Head Attention 구조



자료: "Attention Is All You Need" 논문

Multi-Head Attention의 수식적인 표현은 다음과 같다.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_\text{h})W^O$$

$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$.

이때, d_{model}은 모델에서 사용하는 임베딩 벡터의 차원 수임. 실제 논문에서는 512였음.

d_k는 Attention에서 Key를 선형 변환하여 줄인 차원 수이고, d_v는 Value를 선형 변환하여 줄인 차원 수임.

h는 Multi-Head Attention에서의 multi-head 수, 즉 중첩해서 만들어진 Attention 구조의 수를 말함.

위의 수식을 설명하면, 먼저 h개만큼 각각의 Attention 구조를 만든 다음, 각 Attention의 입력 값에는 단순한 Q, K, V가 아닌 Q, K, V에 각기 다른 가중치 행렬이 곱해진 정보가 들어간다. 그리고, 각 Attention을 출력 값을 concat, 즉 횡으로 붙인 다음에, 여기에 다시 $\boldsymbol{W^O}$의 가중치 행렬을 통과하여 최종적인 종합 결과가 나오도록 했다.

이 Multi-Head Attention에서 학습 가능한 가중치 행렬의 종류에는 $\boldsymbol{W_i^Q, W_i^K, W_i^V, W^O}$의 4종이 있다. 따라서 멀티 헤드 개수를 늘릴수록, 원론적으로는 $\boldsymbol{W_i^Q, W_i^K, W_i^V}$의 가중치 행렬이 그만큼 생겨난다고 할 수 있다.

실제 논문에서는 d_{model}로 512를 사용했다. h(멀티 헤드의 수)는 8을 사용했다. 그리고, d_k, d_v는 64를 사용했다. 이런 실제 논문의 적용 기준(수치)을 바탕으로, 자세히 설명해 보자.

우선 모델은 텍스트 토큰(단어)을 표시할 때 512차원 벡터로 기록한다(임베딩 벡터, 임베딩 레이어). 따라서 n_tokens 개의 토큰(단어)으로 이루어진 한 문장은 (n_tokens, 512)의 행렬로 표현된다.

이 (n_tokens, 512)의 행렬 구조로 이루어진 Q, K, V를 가지고 Attention 계산을 한다고 하면, 우선 모두 (512, 64)의 가중치 행렬이 곱해져 (n, 64) 행렬이 되면서 특정한 의미 공간으로 한번 변형된다. Attention Value의 계산 결과도 (n, 64) 행렬이 된다. 이런 결과 행렬 8개를 옆으로 붙이면, 모양상으로는 다시 (n_tokens, 512)의 행렬이 된다. 여기에 (512, 512) 행렬인 $\boldsymbol{W^O}$를 곱하여 최종적인 (n_tokens, 512) 행렬이 도출된다.

논문에서는 하나의 Multi-Head Attention 레이어 안에 독립적인 8개의 Attention 계산이 이루어지도록 지시하여, 각각의 Attention이 문장 내 단어 간 연관성을 볼 때 서로 다른 관점의 정보를 추출할 수 있도록 만들었다. 이를 통해 더 뛰어난 문맥 파악, 더 뛰어난 문장 번역 기능이 가능해졌다고 할 수 있다.

4. Self-Attention (#4 / 거의 신규 기술)

예를 들어, "인도 카레 전문점에 가다"와 "인도 주행 금지 표지판을 보았다"란 문장이 각각 있다고 하자(임커밋 유튜브 채널에서의 예시). 두 문장에서의 "인도" 단어는 전혀 다른 뜻이다. 사람의 경우는 두 단어가 다른 의미임을 바로 알 수 있다.

사실, 문장 내 한 단어의 정확한 의미는, 문맥 내에서의 관계 즉 문장 내 다른 단어와의 관계를 통해 명확히 결정된다고 할 수 있다. 즉, 첫 번째 문장의 "인도"와 두 번째 문장의 "인도"는 문장 내 다른 단어들의 존재로 인해 그 뜻의 차이가 두드러진다. 컴퓨터가 이런 작업을 할 때 필요한 것이 Self-Attention 알고리즘이다.

Self-Attention 기법은, Attention의 3가지 입력 변수에 모두 자기 자신의 문장을 넣어서, 문장 내 각 단어가 다른 단어들과 어떻게 연관되는지를 계산하고 이를 고려한 문맥 벡터를 도출하는 기법이다.

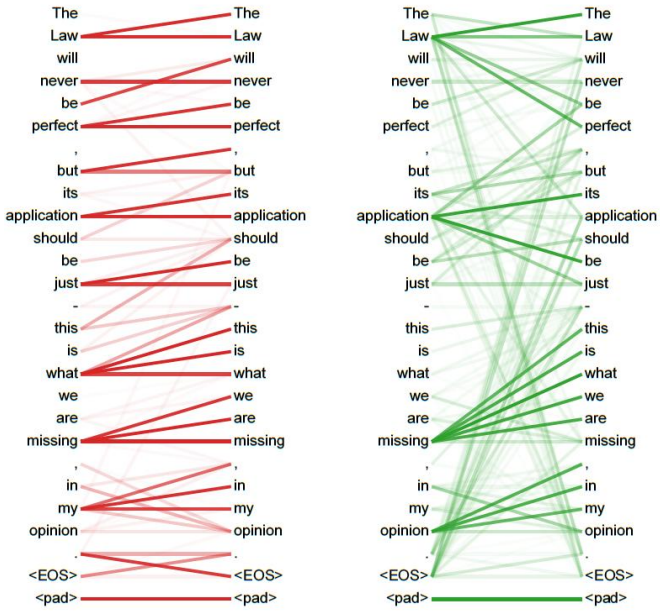
만약 바르게 학습된 Attention 레이어상의 계산이라고 하면, "인도 카레 전문점에 가다"에서의 "인도"와 가장 밀접한 단어를 찾는 단계에서는 그 연관 단어로 "카레", 그다음에 "전문점" 정도로 나올 것이다. "인도 카레 전문점에 가다" 문장에 Self-Attention을 한 번 적용하면, 적용 후 "인도" 토큰 위치의 문맥 벡터는, 카레와 전문점의 의미가 더 가미된 의미 공간으로 움직이게 된다. 이때 물론 단어 간의 연관성을 결정하는 노하우는 Attention 레이어의 가중치 행렬에 저장된다. 이 카레와 전문점의 의미가 가미된 "인도" 벡터는, 번역 task 시 "India"로 맞게 번역될 가능성이 높아진다. "인도 주행 금지 표지판을 보았다" 문장에 Self-Attention을 한 번 적용하면, 적용 후 "인도" 토큰 위치의 문맥 벡터는 주행 혹은 금지의 의미가 가미된 의미 공간으로 움직이게 된다. 이 주행 혹은 금지의 의미가 가미된 "인도" 벡터는, 번역 시 "sidewalk"로 번역될 가능성이 높다. 한 단어와 다른 단어와 얼마나 연관되어 있는지는, Attention 레이어의 Q, K, V에 붙는 가중치 행렬이 학습을 하고 그 노하우를 기록한다.

만약 "인도 카레 전문점에 가던 중 인도 주행 금지 표지판을 보았다"란 문장이 있다고 하자. 단어 임베딩 레이어상에서의 "인도" 단어는 India도 가능하고, sidewalk도 가능한 중간적인 수치 벡터로 표현될 것이다. Self-Attention을 포함하는 신경망 모델이 수많은 문장으로 학습을 하게 되면, Self-Attention의 가중치 행렬이 각 단어 간의 연관성을 잘 학습하게 된다. 따라서, 모델 추론 시 Self-Attention 계산을 거치면 첫 번째 "인도" 벡터는 카레와 전문점 정보가 많이 들어간 문맥 벡터로 바뀌고, 두 번째 "인도" 벡터는 주행과 금지의 정보가 많이 들어간 문맥 벡터로 바뀐다. 이제부터 문맥 정보가 들어간 단어 정보들을 가지고 다양한 작업을 할 수 있게 된다.

즉, Self-Attention은, 각 단어에 대해서 문장 내 다른 단어와의 연관성 정보를 추가한 정보를 만들어서 다음 레이어로 전달하는 역할을 한다. 이를 통해서 정확한 문맥 이해가 필요한 작업들이 가능하도록 만든다.

$$SelfAttention(\boldsymbol{X}) = Attention(\boldsymbol{XW_Q}, \boldsymbol{XW_K}, \boldsymbol{XW_V})$$

Self-Attention 예시



자료: "Attention Is All You Need" 논문, Figure 5

위의 예시를 보면, Self-Attention이 어떻게 문맥 정보를 반영하는지의 예시를 볼 수 있다. Multi-Head Attention 하에서 각 Attention head는 서로 다른 연관성을 추적할 수 있다. 왼쪽의 빨간 색 Attention 그래프는 인접한 단어나 문법적 짝을 이루는 단어 간의 연관성을 나타내고 있다. 오른쪽의 녹색 Attention 그래프는, 장거리의 의미 의존성이 높은 단어 간의 연결 혹은 전역적 문맥을 이해하는 역할을 한다고 볼 수 있다.

5. Masked Attention (#3.2.3 / 신규 기술)

Masked Attention은, 디코더의 Self-Attention 단계에서만 사용되는 특수한 Attention 내 처리 기법을 말한다. Masked Attention은 디코더 안에서 정보의 흐름이 왼쪽의 역방향으로 흐르는 것을 막는 기능을 한다. 예를 들어 영어를 독일어로 번역하는 작업을 한다고 하자. 영어 문장(인코더 문장)은 전체를 처음부터 볼 수 있지만, 새로 만드는 독일어 문장(디코더 문장)은 전체 문장이 당연히 없는 상태에서 시작한다. 독일어로 한 단어씩 작성하면서, 직전까지의 디코더 문장과 전체 인코더 문장을 참고해서 새로운 단어를 만들어내게 된다. 모델이 제대로 학습하려면 이렇게 순차적으로 문장이 생성되는 환경을 그대로 만들어줘야 한다.

일반적인 Attention 계산에서는 모든 토큰 대 모든 토큰의 Attention을 한 번에 계산하는 방식이다. 그대로 사용하면, 현재 시점보다 뒤에 있는 단어들을 참고하는 잘못된 형태가 된다.

"현재 시점보다 뒤의 단어들을 보지 못하게 가리기 위해서" Masked 방식에서는, softmax 계산 직전 단계에서 Attention Score Matrix 상 현재 단어 위치 기준 오른쪽에 있는 값들을 모두 음의 무한대(-∞)로 바꾼다(정확히는 large negative number를 사용함). 이를 통해서, softmax 통과 후 사후 단어 위치의 Attention Weight는 0이 되어, 정보를 참조할 수 없도록 만들어준다.

($\frac{QK^T}{\sqrt{d_k}}$의 직후 및 $softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)$의 직전 단계에서 처리함.)

6. Position-wise Feed-Forward Networks (#3.3 / 기존 기술)

일반적인 용어로 feed-forward network는 순방향 신경망을 의미한다. 하지만, 이 논문에서의 position-wise feed-forward networks는, 전체 구조 중 Attention 절차의 바로 다음에 붙인 특별한 서브 레이어를 뜻한다. 특히 "position-wise"란 용어에 주의할 필요가 있다. 여기서 "position-wise"의 뜻은, "위치(position, 스텝, 혹은 단어)별로 독립적으로 적용되는"의 의미를 가진다.

논문에서의 수식은 다음과 같다.

$$FFN(x) = \max(0, xW_1 + b_1)\, W_2 + b_2$$

a fully connected feed-forward network, which is applied to each position separately and identically."

위의 수식 자체는, 일반적인 데이터 다음에 dense 레이어를 2번 붙인 모양에 해당한다. 그런데, 각 position(단어, 토큰)별로는 각각 계산을 하는데, 이 dense 레이어의 가중치 행렬은 공통으로 사용하겠다는 뜻이다. 이 내용은, 전체 모델에서 사용되는 데이터의 shape를 고려하면 쉽게 이해할 수 있다.

일반적인 기계 번역의 입력 데이터에서 n_tokens는 토큰의 수, dimension은 단어의 임베딩 벡터의 차원이라고 하면, 한 개 샘플 데이터(한 개 문장)의 shape는 (n_tokens, dimension)이다. 한 개 문장은 여러 개의 단어로 이루어져 있기 때문이다. 이에 따라, 번역 문장들을 모두 모든 전체 샘플 데이터는 (n_samples, n_tokens, dimension)의 텐서 형태가 된다.

기계 번역 입력 데이터의 shape: $(n_samples, n_tokens, dimension)$

이때, n_samples: 샘플 수, n_tokens: 토큰 수, dimension: 임베딩 벡터의 차원.

참고로, 논문은 n_tokens의 최대치를 512, dimension을 512로 설정했음.

지난 26/1/2일 자 "퀀트 모델링 A to Z – (6) MLP(Multi-Layer Perceptron) AI 투자 모델" 리포트에서 본, 기본 dense 레이어는 (n_samples, f) shape 데이터를 입력으로 받고, (n_samples, h) shape 데이터를 출력으로 보내는 구조였다. 이 dense 레이어의 작동은, 사실은 입력 데이터 구조에 따라서 다른 방식으로 움직인다.

dense 레이어 작동법:

1) dense 레이어의 가중치 행렬 shape가 (f, h)이고, 입력 데이터 shape가 (n_samples, f)면, dense 레이어 출력의 shape는 (n_samples, h)가 된다.

2) dense 레이어의 가중치 행렬 shape가 (f, h)이고, 입력 데이터 shape가 (n_samples, n_tokens, f)면, dense 레이어 출력의 shape는 (n_samples, n_tokens, h)가 된다.

위의 2개 케이스가 다르게 움직인다.

keras의 dense 레이어의 작동은 입력 데이터 구조에 따라서, 입력 데이터가 2차원 행렬일 때는 레이어의 출력이 2차원 행렬이 되고, 입력 데이터가 3차원 이상의 텐서면 입력의 마지막 2개 축을 행렬로 보고 계산한 다음 텐서로 출력을 하게 된다. 이는, numpy의 broadcasting 작동법에서, 행렬 곱셈 시 마지막 두 축을 행렬로 보고, 그 앞의 축들은 batch 차원으로 broadcasting 하는 것과 같은 원리다.

이들 작동법에서 1번 케이스가, 일반적인 MLP 즉 dense 레이어의 사용법이다. 2번 케이스는 사실은, kernel size=1인 Conv1D 컨볼루션 신경망을 쓰는 것과 같은 의미이다.

keras.layers.Dense의 설명문

Dense implements the operation: output = activation(dot(input, kernel) + bias) where activation is the element-wise activation function passed as the activation argument, kernel is a weights matrix created by the layer, and bias is a bias vector created by the layer (only applicable if use_bias is True).

Note: If the input to the layer has a rank greater than 2, Dense computes the dot product between the inputs and the kernel along the last axis of the inputs and axis 0 of the kernel (using tf.tensordot). For example, if input has dimensions (batch_size, d0, d1), then we create a kernel with shape (d1, units), and the kernel operates along axis 2 of the input, on every sub-tensor of shape (1, 1, d1) (there are batch_size * d0 such sub-tensors). The output in this case will have shape (batch_size, d0, units).

입력 데이터가 3차원 텐서인 이 "2번째 작동법"의 처리를 보면, 레이어의 가중치 행렬은 (f, h) shape를 가지고 있어서, 입력 벡터(f 차원)의 정보를 출력 벡터(h 차원)로 일대일로 변환하는 정보를 담게 된다. 그리고, 이 가중치 행렬의 학습에는 모든 샘플의(n_samples), 모든 토큰(n_tokens)이 다 사용된다. 이 변환 레이어는, 여러 토큰을 동시에 참조하는 방식이 아니고, 한 개 토큰(혹은 벡터)만을 입력으로 받아서, 다음 층으로 한 개 벡터로만 출력하는 일대일 변환 작업을 한다. 이런 작동 방식을 "position-wise"라고 한다.

position-wise: (n_samples, n_tokens, f) → (n_samples, n_tokens, h)로 변환

사실, 이 하나의 연결 층은, 정확하게 "kernel size=1의 Conv1D" 형태 컨볼루션 레이어와 내용상에서 완전히 동일하다. 입력 데이터가 (n_samples, n_tokens, f)일 때, Conv1D에서 kernel size=1이고 필터 수가 h인 "Conv1D 레이어"를 가정해 보자. Kernel size가 1이기 때문에, 작업 시 "한 개 position(timestep 혹은 length)"만을 사용하고, 입력 데이터에 f개 채널이 있기 때문에, 한 개 position 상의 전체 f개 채널의 선형 조합을 통해서 필터를 h개 만들어낸다. 즉, (n_samples, n_tokens, f) → (n_samples, n_tokens, h) 변환이다. 이는 앞에서 본 "position-wise" 방식도 동일하다. 즉, "3차원 텐서 데이터를 입력으로 하는 dense 레이어"는 kernel size=1의 Conv1D 레이어와 동일한 기능을 하는 것을 알 수 있다.

논문에서 말하는 Position-wise FFN층은, 정확히"kernel size=1의 Conv1D 레이어" 2개를 연속해 붙이는 레이어와 같다. (kernel size=1의 Conv1D는 1x1 convolution이라고도 함. 1x1 convolution은 공간 방향으로는 변경하지 않고 채널 방향으로만 mixing 하므로 동일한 연산 구조임.)

이 신경망에서는 해당 층 통과 시 한 토큰씩 독립적으로 처리가 진행되며, 다른 토큰과 정보가 섞이는 변화는 나타나지 않는다. 이 논문에서 이 FFN 층의 내부 차원은 d_{ff}=2048을 사용한다. 따라서, 이 FFN의 첫 번째 레이어

는 kernel size=1이고 필터 수가 2,048인 Conv1D 층이며, 두 번째 레이어는 kernel size=1이고 필터 수가 512인 Conv1D 층으로 볼 수 있다. 혹은, (n_samples, n_tokens, 512)인 데이터를 (n_samples, n_tokens, 2048)로 변환하는 dense 층과, (n_samples, n_tokens, 2048)인 데이터를 (n_samples, n_tokens, 512)로 변환하는 dense 층을 붙였다고 말할 수 있다.

데이터의 변화를 보면, 처음의 문장 데이터는 (n_tokens, 512) 행렬이었다가, FFN의 첫 번째 층을 지나면 (n_tokens,2048)로 채널(정보)이 확장된다. 그리고, FNN의 두 번째 층을 지나면서 다시 (n_tokens,512) 행렬로 돌아온다. 이런 연산과 내부 가중치 행렬의 학습을 통해서, 전체 모델이 더 많은 지식을 저장하여 기계 번역을 원활하게 만들어준다. 이처럼 차원을 한 번 바꿨다가(늘리거나 줄이거나) 다시 원래 차원으로 되돌리는 MLP 구조를 Bottleneck(병목) MLP라고 한다. 중간의 차원을 늘리는 Expansion Bottleneck의 경우에는 데이터의 표현력이 증가하고, 더 복잡한 feature 조합이 가능하고, 비선형 표현을 강화하는 장점이 나타나게 된다.

Transformer 모델 내의 역할을 보면, Attention 레이어 가중치 행렬에는 단어 간의 관계 및 문맥 파악과 관련한 정보가 주로 많이 저장된다. 이와 달리 FFN의 가중치 행렬에는 개별 단어의 의미 개념 감지 및 실제 텍스트로의 표현 변환을 담당한다고 볼 수 있다. Transformer 모델 내에서, 사실은 이 FFN 레이어가 Attention 레이어보다 더 많은 파라미터를 보유하고 있다. 즉, 이 FFN 레이어가 모델의 실제 성능에 큰 영향을 준다고 할 수 있다.

7. Sinusoidal Positional Encoding (#3.5 / 신규 기술)

Positional Encoding은 Transformer 모델에서 Attention 기술 못지않게 중요한 역할을 하는 알고리즘이다. 지금까지 본 것처럼 Transformer 모델은, RNN 같은 순환 신경망과 달리 문장의 모든 토큰(단어)을 동시에 처리한다. 이때 단어의 임베딩 벡터 자체에는 이 단어가 문장 내에서 앞쪽에 있는 단어인지, 문장 내 뒤쪽에 있는 단어인지 구별할 수 없다. 문장 내에서 단어의 순서, 위치는 의미를 결정하는 데 핵심적인 역할을 할 수도 있다(예로, "A가 B를 사랑한다"와 "B가 A를 사랑한다"). 따라서, 문장 내의 단어 위치를 입력 정보에 추가하는 작업이 꼭 필요하다.

Transformer의 Positional Encoding은, 아래의 수식을 가지고 각 단어의 위치 정보를 수치로 변환하여 단어의 원래 임베딩 벡터에 더해준다. 이때 sin, cos 함수를 사용하므로, 이를 Sinusoidal Positional Encoding 방식이라고도 칭한다.

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

이때, pos는 토큰(단어)의 문장 내 위치, i는 임베딩 벡터 내의 차원 인덱스임.

원논문은 d_{model}=512여서, 텍스트 토큰을 512차원의 임베딩 벡터로 표현한다. Positional Encoding 시에는, 각 토큰의 위치와 임베딩 벡터의 인덱스 위치에 맞게 계산된 "PE 수치"가 해당 토큰의 각 벡터 원소에 더해지면서, 임베딩 벡터의 정보가 변형이 되는 구조다.

이때 실제로 더해지는 PE 값이 어떻게 바뀌는지를 예시를 보자. 모델에서처럼 d_{model} =512로 정하고, position=0번 토큰부터 19번 토큰까지 더해지는 Positional Encoding 벡터 수치 일부를 가져오면 다음과 같다.

position=0번 토큰부터 19번 토큰까지 더해지는 Positional Encoding 벡터 수치 (일부)

pos	dim_0	dim_1	dim_2	dim_3	dim_4	dim_5	dim_6	dim_7	...	dim_54	dim_55	...	dim_256	dim_257	...	dim_510	dim_511
0	0.000000	1.000000	0.000000	1.000000	0.000000	1.000000	0.000000	1.000000		0.000000	1.000000		0.000000	1.000000		0.000000	1.000000
1	0.841471	0.540302	0.821856	0.569695	0.801962	0.597375	0.781887	0.623420		0.369575	0.929201		0.010000	0.999950		0.000104	1.000000
2	0.909297	-0.416147	0.936415	-0.350895	0.958144	-0.286285	0.974888	-0.222695		0.686819	0.726829		0.019999	0.999800		0.000207	1.000000
3	0.141120	-0.989992	0.245085	-0.969501	0.342782	-0.939415	0.433643	-0.901085		0.906810	0.421539		0.029996	0.999550		0.000311	1.000000
4	-0.756802	-0.653644	-0.657167	-0.753745	-0.548606	-0.836081	-0.434205	-0.900814		0.998399	0.056560		0.039989	0.999200		0.000415	1.000000
5	-0.958924	0.283662	-0.993855	0.110692	-0.998229	-0.059494	-0.975027	-0.222086		0.948617	-0.316428		0.049979	0.998750		0.000518	1.000000
6	-0.279415	0.960170	-0.475221	0.879866	-0.644029	0.765001	-0.781498	0.623908		0.764512	-0.644610		0.059964	0.998201		0.000622	1.000000
7	0.656987	0.753902	0.452392	0.891819	0.228775	0.973479	0.000625	1.000000		0.472153	-0.881516		0.069943	0.997551		0.000726	1.000000
8	0.989358	-0.145500	0.990673	0.136263	0.917358	0.398064	0.782276	0.622932		0.112939	-0.993602		0.079915	0.996802		0.000829	1.000000
9	0.412118	-0.911130	0.676370	-0.736562	0.867239	-0.497892	0.974749	-0.223304		-0.262267	-0.964995		0.089879	0.995953		0.000933	1.000000
10	-0.544021	-0.839072	-0.220023	-0.975495	0.118776	-0.992921	0.433080	-0.901356		-0.600337	-0.799747		0.099833	0.995004		0.001037	0.999999
11	-0.999990	0.004426	-0.927062	-0.374907	-0.725331	-0.688401	-0.434768	-0.900543		-0.853400	-0.521256		0.109778	0.993956		0.001140	0.999999
12	-0.536573	0.843854	-0.836262	0.548329	-0.985366	0.170454	-0.975166	-0.221477		-0.985624	-0.168956		0.119712	0.992809		0.001244	0.999999
13	0.420167	0.907447	-0.025767	0.999668	-0.451936	0.892051	-0.781108	0.624396		-0.978284	0.207267		0.129634	0.991562		0.001348	0.999999
14	0.990607	0.136737	0.806904	0.590682	0.445415	0.895324	0.001249	0.999999		-0.832422	0.554142		0.139543	0.990216		0.001451	0.999999
15	0.650288	-0.759688	0.945145	-0.326650	0.984096	0.177639	0.782665	0.622443		-0.568690	0.822552		0.149438	0.988771		0.001555	0.999999
16	-0.287903	-0.957659	0.269985	-0.962865	0.730334	-0.683090	0.974609	-0.223913		-0.224433	0.974490		0.159318	0.987227		0.001659	0.999999
17	-0.961397	-0.275163	-0.637527	-0.770428	-0.111529	-0.993761	0.432517	-0.901626		0.151604	0.988441		0.169182	0.985585		0.001762	0.999998
18	-0.750987	0.660317	-0.996377	0.085047	-0.863583	-0.504206	-0.435330	-0.900271		0.506174	0.862432		0.179030	0.983844		0.001866	0.999998
19	0.149877	0.988705	-0.497735	0.867329	-0.920238	0.391360	-0.975304	-0.220868		0.789070	0.614303		0.188859	0.982004		0.001970	0.999998

비유해서 이야기하면, 이 Positional Encoding 정보 혹은 수치는, 시계가 초침, 분침, 시침을 통해서 하루 중의 절대 시간을 나타내는 것과 유사한 원리다. 벡터 원소를 2개씩 짝지어서, 2개 수치 쌍이 시계 침의 (x, y) 좌표를 나타내는 방식이다. 좀 더 정확히 비유하자면, 마이크로 초침, 밀리 초침, 초침, 분침, 시침, Weekly 침 등 서로 다른 주기를 가지는 시계 침 총 256개(2개씩 짝)를 사용해, 위치 정보를 기록하는 방식이다.

PE 벡터 수치를 각 인덱스별로 나눠 보면, 우선 index=0과 1의 수치 쌍은 $sin(pos/10000^{0/512})$, $cos(pos/10000^{0/512})$이 되면서 단어 위치(pos)가 하나씩 올라갈 때 수치가 매우 빨리 변화한다(앞의 표에서의 수치/색상 변화 참고).

그러나, index=510과 511의 수치 쌍은 $sin(pos/10000^{510/512})$, $cos(pos/10000^{510/512})$이 되면서 상당히 주기가 긴 파동을 나타낸다. 이에 따라 단어 위치가 하나씩 올라갈 때에 수치가 매우 느리게 변화한다. 실제로 index=511 위치 기준에서는 문장 내에서 약 62,832번째 위치의 단어가 되어야만 같은 수치를 가질 수 있게 된다. 즉, 62,832번째 단어가 있어야만 한 바퀴를 도는 시계 침의 (x, y) 좌표 값이다.

이렇게 위치 벡터의 각 인덱스마다 서로 다른 주기로 파동 하는 sin, cos 함수를 설정함으로써, 문장 내 n번째 단어라고 하는 유니크한 정보를, 수치가 발산하지 않으면서도, 서로 다른 수치들로 잘 표현할 수 있게 해준다. 이렇게 설정된 index=0 ~ index=511의 수치들을 해당 단어의 원래 임베딩 벡터에 더해 주어서 위치 정보를 심게 된다.

한편, 이 Positional Encoding 벡터에서의 뒤쪽 차원 수치는 매우 느리게 변하면서 "이 단어는 문장에서 앞부분인가, 뒷부분인가"의 거시적인 위치 정보를 제공한다. 이 벡터의 앞쪽 차원 수치는 매우 빠르게 변하면서, 예를 들어 두 단어가 있을 때 이 단어들 위치가 바로 옆인지, 몇 칸 떨어져 있는지를 잘 구별하는 성질을 가지고 있다.

또한 이 Positional Encoding 방식은, 두 단어 간의 상대적인 위치 관계도 어느 정도 관리해 주는 장점이 있다.

Positional Encoding 벡터에서 $2i$ 위치와 $2i+1$ 위치의 수치는 같은 주기를 가지는 sin 함수와 cos 함수에 해당한다. 즉, $2i$, $2i+1$번째 수치를 x, y축에 놓으면, 지름이 1인 원 위에 계속해서 좌표가 찍히게 된다.

이때의 사인과 코사인 수치는 원 위에서 회전하는 위치를 나타내는 역할을 한다. 삼각함수의 덧셈 정리로 인해, 극좌표계 상의 $a+b$ 각도의 위치를 직교좌표계로 표시한 $\left(\cos(a+b), \sin(a+b)\right)$ 는, a 각 위치 $\left(\cos(a), \sin(a)\right)$에서 b각만큼 회전한 위치에 해당한다. 이와 동일하게 $(\sin x, \cos x)$ 좌표는 12시에서 출발해서 시계 방향으로 x 각만큼 움직인 위치를 뜻한다고 할 수 있다.

예를 들어 "$pos+k$"번째 단어의 Positional Encoding 벡터의 $(2i, 2i+1)$ 인덱스 쌍은 다음과 같이 표현된다.

$$(sin\big((pos+k)*N\big), \cos\left((pos+k)*N\right))$$

삼각함수의 덧셈정리로 인해서, 이는 "pos"번째 단어의 PE 벡터 $(2i, 2i+1)$ 인덱스 쌍에서, k번만큼의 단위 각도를 옮긴 위치에 해당한다. 즉, 문장 내 단어의 위치가 "한" 칸씩 증가할수록, Positional Encoding 벡터에서는 256개의 톱니가, 톱니 간에는 서로 다른 각도지만(저 인덱스에서는 빠르게, 고 인덱스에서는 느리게) 한 칸의 움직임은 항상 동일한 폭이 되도록, 톱니가 맞물리면서 회전하는 구조가 된다.

이에 따라, 이 Positional Encoding 벡터는 두 단어 간의 위치 차이도 보여줄 수 있고, 두 위치 간 거리 차이의 크기도 어느 정도 보존하는 역할을 하게 된다.

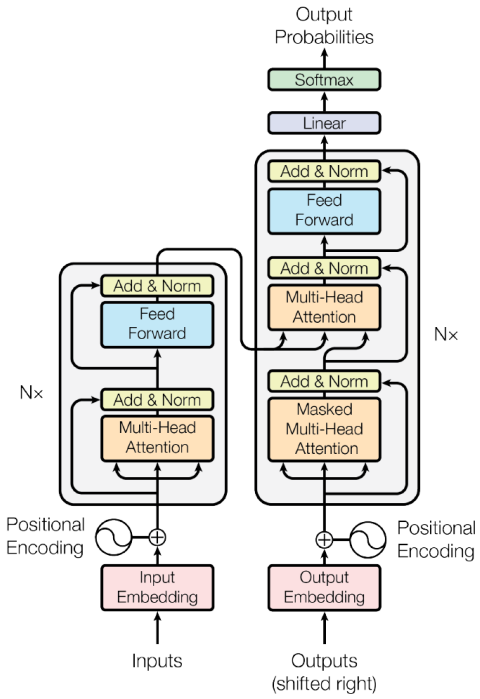
한편으로, Positional Encoding 벡터를 토큰의 원래 임베딩 벡터에 단순히 더하는 작업을 했을 때, 그 정보가 유지가 되냐는 점이 궁금할 수 있다. 우선 위치 벡터를 "더하는" 이유는, 위치 정보를 벡터 차원을 늘리는 형태로 결합하면 계산 복잡도가 증가하여 효율성이 떨어지기 때문이다. "차원의 효율성" 때문에 결합이 아닌 더하기 방식을 사용한다. 그리고, 임베딩 벡터의 차원 자체가 큰 편이기 때문에, 더하기 방식을 사용해도 고차원 공간에서 두 벡터가 각자 어느 정도 고유의 방향성을 유지할 수 있어서, 이런 더하기 연산을 사용한다.

이후, 이 Positional Encoding 기법은 계속 발전하여, 최근에는 각도 결정 방식은 동일하지만 위치 정보의 삽입을 Attention 레이어와 연계해서 진행하는 RoPE(Rotary Positional Embedding) 방식이 많이 사용되고 있다.

8. 정리

지금까지 정리한 모든 내용을 가지고, Transformer의 구조를 다시 한번 확인해 보자.

Transformer 아키텍처의 전체 구조도



참고: 논문에서는 레이어 반복 횟수 N=6을 적용함
자료: "Attention Is All You Need" 논문

모델 구조도를 보면, 인코더 파트에서 맨 처음에 텍스트 데이터를 임베딩 벡터로 변환한다. 그다음, Positional Encoding을 통해서, 각 문장의 토큰에 위치 정보가 포함된다. 그리고, 6번의 독립적인 레이어를 진행하는데, 각 레이어 안에서는 Attention 서브 레이어와 Position-wise FFN 서브 레이어가 순차적으로 진행된다. 이 Attention과 FFN 서브 레이어 모두가 Residual Connection 및 레이어 정규화를 사용한다.

디코더 파트에서도 처음에 텍스트 데이터가 임베딩 벡터로 변환하고, 그다음 Positional Encoding으로 토큰에 위치 정보가 포함된다. 디코더도 6번의 독립적인 레이어를 진행하는데, 각 레이어 안에서는 Masked Multi-Head Self Attention 서브 레이어, 인코더의 정보를 같이 사용하는 Multi-Head Cross Attention 서브 레이어, Position-wise FFN 서브 레이어가 순차적으로 진행된다. 각 서브 레이어는 모두 Residual Connection 및 레이어 정규화를 사용한다. 이 레이어의 전체 진행이 끝나면, 마지막에는 linear 층 통과와 Softmax 함수를 통해서, 최적 단어를 출력하는 작업을 진행한다. 여기까지가 Transformer 모델의 전체 절차다.

Transformer 전체 구조를 생각해 보면, Transformer는 Self-Attention과 Multi-Head Attention을 반복 처리하면서, 토큰들이 문장 전체의 문맥을 파악하면서 의미를 계속 업데이트하여 정확한 기능을 수행하는 시스템이라고 할 수 있다. 이런 혁신적인 기술들을 접목함으로써, Transformer는 발표 당시 타 모델을 압도하는 성능을 증명한 바 있다(논문 내의 Table 2. BLEU Score 비교표 참고). 그리고, 이후로 진화한 수많은 LLM 열매의 뿌리 역할을 했다.

III. Transformer 사용법

MultiHeadAttention 레이어

Transformer 모델과 관련해서, keras 라이브러리에서는 해당 모델의 핵심 요소인 MultiHeadAttention을 구현한 레이어 클래스를 제공한다. 이 MultiHeadAttention 클래스를 사용하면, Multi Head Cross Attention과 Multi Head Self-Attention에 모두 쓸 수 있다.

해당 클래스의 선언 문법은 다음과 같다.

```
keras.layers.MultiHeadAttention(
    num_heads,
    key_dim,
    value_dim=None,
    output_shape=None,
***
)
```

num_heads 파라미터는, 생성할 Attention의 멀티 헤드 개수를 의미한다(논문에서는 8을 사용함). key_dim 파라미터는 멀티 헤드 구조에서 query와 key에 붙는 가중치 행렬의 차원이다(논문에서는 임베딩 벡터 차원이 512였고, 이 key_dim에는 64를 사용함). value_dim은 멀티 헤드에서 value에 붙는 가중치 행렬의 차원이다. None으로 지정하면 key_dim과 같은 값으로 설정된다. output_shape는 레이어 output의 shape를 변경할 때 사용한다. 기본 None으로 지정하면, 입력된 query 데이터와 동일한 형태로 output이 나온다. 만약 shape를 변경하려고 하면, 출력 tensor 모양에서 batch와 sequence 차원을 뺀 마지막 차원 수를 입력하면 된다.

Call(실행) 인자

query: query 텐서 데이터
value: value 텐서 데이터
key: key 텐서 데이터(optional). 생략하면 value 텐서가 key로도 사용됨(보통 생략)
use_causal_mask: 디폴트는 False임. True면, Masked Attention 방식으로 작용함
…

MultiHeadAttention의 실행 시의 주요 인자는 위의 표와 같다. 예를 들어, use_causal_mask 파라미터의 경우 True로 지정하면, Masked Attention 방식으로 작동한다. MultiHeadAttention(X, X, X)의 코드를 실행하면, 이는 Self-Attention을 실행하는 것이다.

AdditiveAttention, Attention 레이어

한편, keras에서는 Transformer 모델의 MultiHeadAttention 레이어뿐 아니라, 초기 Attention 모델에 대응되는 레이어도 제공하고 있다(AdditiveAttention, Attention 레이어). Keras.layers.AdditiveAttention 레이어는 Bahdanau의 덧셈 방식 Attention을 구현한 것이다. Keras.layers.Attention 레이어는 Luong의 dot-product Attention을 구현한 것이다. 이때 주의할 것은, AdditiveAttention과 Attention 레이어 모두 가중치 행렬이 내부에 존재하지 않는다는 점이다. Bahdanau Attention은 원래 내부에 가중치 행렬이 있는 형태지만, keras에서는

선형 변환이 끝난 데이터가 입력으로 들어온다고 가정하여 레이어 내부의 선형 변환을 삭제하는 방식으로 만들었다. Luong Attention은 원래부터 Attention 계산 내부에 가중치 행렬이 없는 형태이므로, Attention 레이어 클래스에서도 내부 가중치 행렬이 없다. 따라서, 이 두 개 레이어를 사용할 때는, 수동으로 직전에 가중치 행렬을 추가하거나, LSTM 등을 활용하는 방식을 써야 함에 주의해야 한다.

TransformerBlock layer 만들기

Transformer 아키텍처에서 큰 블록 레이어 – "MultiHeadAttention 실행과 그 이후의 ResNet 및 LayerNorm 연결, 그리고 Position-wise FFN 실행과 그 이후의 ResNet 및 LayerNorm 연결하는 전체 레이어"는 자주 사용될 가능성이 높다. keras 사이트에서 제공하는 Transformer의 큰 블록 레이어의 작성 코드 예시는 다음과 같다.

Transformer 블록 레이어의 예시 코드

```
from keras import layers

# 트랜스포머 블록 클래스 선언(Self-Attention 기준)
class TransformerBlock(layers.Layer):

    # 초기화: embed_dim=임베딩 차원, num_heads=어텐션 헤드 수, ff_dim=FFN 은닉층 크기, rate=드롭아웃 비율
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):
        super().__init__()
        self.att = keras.layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim    # num_heads개의 헤드, 각 헤드의 key 차원 = embed_dim
        )                                             # 멀티헤드 셀프 어텐션 레이어 준비
        self.ffn = keras.Sequential(
            [
                keras.layers.Dense(ff_dim, activation="relu"),    # FFN 첫 번째 층: ff_dim으로 확장 후 ReLU
                keras.layers.Dense(embed_dim),                    # FFN 두 번째 층: 다시 embed_dim으로 축소
            ]
        )                                                         # Position-wise FFN 준비
        self.layernorm1 = keras.layers.LayerNormalization(epsilon=1e-6)            # 첫째 LayerNorm 준비
        self.layernorm2 = keras.layers.LayerNormalization(epsilon=1e-6)            # 둘째 LayerNorm 준비
        self.dropout1 = keras.layers.Dropout(rate)                                 # 첫째 드롭아웃 준비
        self.dropout2 = keras.layers.Dropout(rate)                                 # 둘째 드롭아웃 준비


    # 순전파 메서드: inputs=입력 텐서, training=학습/추론 모드 구분
    def call(self, inputs, training=False):
        # 셀프 어텐션: query/key/value=inputs
        attn_output = self.att(inputs, inputs)                                  # 셀프 어텐션 실행
        attn_output = self.dropout1(attn_output, training=training)             # 드롭아웃 적용
        out1 = self.layernorm1(inputs + attn_output)                            # 잔차 연결 후 Layer Norm
        ffn_output = self.ffn(out1)                                             # FFN에 통과
        ffn_output = self.dropout2(ffn_output, training=training)               # 드롭아웃 적용
        return self.layernorm2(out1 + ffn_output)                               # 잔차 연결 후 Layer Norm
```

자료: https://keras.io/examples/nlp/ner_transformers/

각 줄 코드에 대한 설명은, 옆에 추가한 주석의 내용을 같이 확인하면 이해 가능하다.

IV. Transformer의 금융 분석 접목

뛰어난 성능을 가진 Transformer 모델 및 이의 중요 알고리즘인 Attention 알고리즘 등을, 금융시장의 분석 및 투자에서 활용하려는 시도 또한 늘고 있다. 여기서는 Transformer 등을 주식 시장 투자 등에 활용한 주요 학술 논문을 정리하여, 해당 내용의 요약 및 평가를 진행하였다.

Wang, Chaojie, et al. "Stock market index prediction using deep Transformer model." Expert Systems with Applications 208 (2022): 118128.

이 논문은 S&P 500, HSI, Nikkei 225, CSI 300 지수에 대해서 Transformer 모델로 지수를 예측하고, 그 예측값에 기반한 롱/숏 전략 성과를 보여준다. 전체 샘플 기간은 2010년부터 2020년까지이고, timesteps는 직전 9일치를 넣는 롤링 윈도우 방식 데이터다. 입력 데이터에서 feature로는 지수값 하나만 사용하는 매우 간단한 방식이다. 논문 결론은 Transformer 모델 기반 전략이 CNN 혹은 LSTM보다 훌륭하다고 주장하고 있다. 단, Transformer 모델 구조를 그대로 가져가기 위해, 1개 feature를 선형 변환으로 32차원 임베딩 벡터로 늘린 점은 문제가 있는 편이다. 전체적으로 과거 주가라는 단편적인 데이터만 쓰고, 매우 짧은 timesteps를 사용했다는 점에서 실제 유용성은 떨어진다고 하겠다.

Kisiel, Damian, and Denise Gorse. "Portfolio Transformer for Attention-Based Asset Allocation." arXiv preprint arXiv:2206.03246 (2022).

기본적으로 자산 배분 방식은 수익률 예측과 포트폴리오 최적화의 두 단계로 나뉘어 있는 형태라서 둘 간의 매끄러운 연결 또한 어려운 임무에 해당한다. 이 논문은 이를 극복하기 위해 Portfolio Transformer(PT)라는 새로운 end-to-end 딥러닝 아키텍처를 제안한다. PT는 개별 자산 수익률을 개별적으로 예측하는 대신, loss function을 Sharpe ratio로 지정하여, 바로 최적화된 자산 비중을 출력하는 방식을 사용했다. 각 자산 배분은 ETFs, Commodities, Stocks를 대상으로 하고, ETFs는 4개 ETF, Commodities는 24개 상품 선물, Stocks는 500개 주식을 자산 배분 대상으로 했고, 입력 데이터도 이들의 수익률 수치다. 인코더 데이터의 timestep은 t-99 ~ t-50, 디코더 데이터 timestep은 t-49 ~ t까지로, 이분법적 입력 구조를 가지고 있다. loss function에 최종 목적 함수인 Sharpe ratio를 바로 지정한 것이 눈여겨볼 만한 지점이다.

Li, Shuozhe, Zachery B. Schulwolf, and Risto Miikkulainen. "Transformer Based Time-Series Forecasting For Stock." arXiv preprint arXiv:2502.09625 (2025).

이 논문은, 주가 예측을 위해 Stockformer라는 변형된 트랜스포머 모델을 제시한다. 여러 종목의 주가를 예측하는 다변량 예측 방식이고, 여러 종목의 다기간 수익률을 입력 데이터로 한다. 비교적 최근의 Informer 모델에서 차용한 ProbSparse Attention과 Self-Attention Distilling 기술을 적용하여, 시간 및 메모리 복잡도를 줄이는 방식을 썼다. 손실 함수에서도 MSE나 MAE 외에 실제 투자 수익률과 관련된 맞춤 함수를 추가 설정한 것도 특징이다. 앞서 자산 배분 전략과 유사하게, 종목의 수익률 간 연관, 영향도를 Attention 알고리즘으로 탐지하는 방식이라고 할 수 있다.

Lin, Hualing, and Qiubi Sun. "Financial Volatility Forecasting: A Sparse Multi-Head Attention Neural Network." Information 12.10 (2021): 419.

이 논문은, Transformer 모델 등에서 차용한 Sparse Multi-Head Attention을 사용해, 자산 가격의 변동성을 예측하는 모델을 제시한다. Sparse Multi-Head Attention은 모든 데이터 대신 상관관계가 높은 주요 쿼리들만 선택하여 계산 복잡도를 줄이는 장점 등이 있다. 주식, 원유, 금을 대상으로 삼고, 가격 데이터를 입력 데이터로 지정하고, 다음 기의 변동성을 출력하는 형태를 가지고 있다. 기존 종가 데이터뿐 아니라 OHLCV의 추가 가격 데이터를 보조 금융 변수로 사용한 것도 장점에 해당한다. 변동성 예측 모델의 경우 보통 가격 지표만을 가지고 예측하는 것이 일반적인 형태이므로, 해당 모델의 데이터 사용 측면을 보자면 크게 약점이 보이지 않는 편이다.

이상으로 금융시장 분석과 Transformer/Attention 모델을 접목한 여러 논문을 확인해 보았다. 논문에 따라 loss function의 맞춤 설정 등 중요한 시도들이 있음을 확인할 수 있다. 한편으로는 금융 도메인의 특징과 데이터에 맞는 형태로 전체 모델의 구조를 최적화하는 변화가 차후에는 더 많아질 것으로 기대한다.

정리

금융시장의 발전 양상을 보면 그 시대의 최신 과학 기술과의 접목을 통해서 퀀텀 점프가 한 번씩 나타났었다고 볼 수 있다. 최근 딥러닝 및 AI의 급격한 발전은 전 세계의 모든 영역에서 큰 영향을 끼치고 있는 상황이다. 이번 Transformer 모델과 같은 최신 AI 기술의 접목이, 금융 투자에 있어서 새로운 기회를 만들 수 있을 것으로 예상한다.

참고문헌

Vaswani, Ashish, et al. "Attention Is All You Need." Advances in Neural Information Processing Systems, vol. 30, 2017.

Special Thanks To. 임커밋 유튜브 채널(www.youtube.com/@임커밋),
임커밋 오픈카톡방(https://open.kakao.com/o/gdjHmPzh)

삼성 All That Quant

[팩터 모델 실전 전략 및 이론]

25/10/27 "영업이익 증가율 모멘텀 팩터 - 신규 추천 전략" - High return, Low correlation의 고효율 팩터전략 신규 개발 및 제시
25/8/19 "Long-only 전략 상세 분석법 (롱 팩터 전략 시리즈 #2)" - 효율적 Long-biased 전략을 위한 프로세스와 추천 전략 제시
25/7/21 "Long-only 전략과 비대칭 알파 (롱 팩터 전략 시리즈 #1)" - 코스피 5000 시대를 대비하는, long-biased 전략 집중 분석
24/11/20 "연구개발 팩터 투자전략" - R&D에 기반한 신규 팩터 전략을 개발하고, 이의 특징과 활용 방법 등을 제시함
24/7/2 "연기금 투자자를 위한 장기 퀀트 전략" – 투자 horizon을 장기로 가져갔을 때의 팩터 투자 전략의 변화를 상세하게 비교 정리함
23/11/30 "퀀트 전략에서의 업종 특성 고려, 2편 - 업종 대비 상대 밸류에이션 팩터의 개선 전략" – 상대 밸류 팩터에서의 업종 특성 고려 전략 제시
23/9/4 "퀀트 전략에서의 업종 특성 고려, 1편 – 저P/B 팩터의 업종 조정 전략" – 한국 업종의 특성을 감안한 수정된 밸류에이션 전략 제시
23/7/26 "Data-driven Quant Model 2 - #연속 국면 #머신러닝 #부스팅 기법 #동적 팩터 모델" – 부스팅 알고리즘 기반의 팩터 로테이션 모델 제시
23/5/17 "퀄리티 팩터는 정말 퀄리티 있는 전략인가" - 퀄리티 팩터가 가지는 고유 특성과 세부 전략 간의 차이점을 비교 정리
23/1/4 "포트 전략 - 롱/숏 시그널로 롱 포트폴리오 구성하기" - 롱/숏 전략을 Long-only 포트로 전환하는 효율적인 알고리즘을 신규 제시
22/11/21 "Data-driven Quant Model – 머신러닝 기법으로 시장의 파도에 올라타기" – 머신러닝 기법에 의한 데이터 기반의 팩터 로테이션 모델 제시
22/8/17 "변동성 구간에서의 주식 스타일 특징 - 낙폭 과대주 팩터 활용법" - 팩터 투자 틀 안에서 낙폭 과대주 팩터의 필요성 입증과 활용법 제시
22/5/25 "경기 국면에 따른 리스크 팩터 배분 전략" - 리스크 팩터 전반에 대한 설명과 함께 팩터와 경기 국면을 결합한 투자 전략을 제시
21/2/24 "이익 모멘텀의 차별화" – 종목 특성을 감안한, 변동성 조정 이익 모멘텀 팩터 전략을 제시
21/1/11 "주식 스타일 Fact Book - 주식 스타일 및 매크로 변수 상관성 팩트 체크" – 주식 스타일 분석 체계의 정리 및 매크로 변수 연결
19/4/15 "팩터 모델의 구조와 해설 - Factor Models for Asset Returns" – 팩터 모델 이론에 대한 상세 설명 자료
18/7/3 "Fama-French 3 팩터 모델 – 후편" - Fama & French의 3팩터 모델 1993년 논문에 대한 전문 번역과 한국 시장 분석 추가 자료
18/5/8 "Fama-French 3 팩터 모델 – 전편" – Fama & French의 3팩터 모델 1992년 논문에 대한 전문 번역과 한국 시장 분석 추가 자료
15/10/20 "가치주 투자와 금리의 연관성" – 금리와 가치/성장 스타일의 인과 분석을, Equity Duration 개념을 활용하여 국내 최초로 제시
14/11/24 "투자 Horizon에 따른 퀀트전략 변화 - Advanced Factor Book" – 1M ~ 6M의 투자 기간별로 팩터 모델 백테스팅 분석
13/4/23 "매크로 기반 Dynamic Quant 모델" – 매크로 국면 판단에 기반한 동적 팩터 투자 모델 정립
12/2/27 "스타일 로테이션의 허와 실" - 스타일 로테이션 전략의 형태로서 모멘텀 방식, 리버전(컨트래리안) 방식, Asness 방식을 각기 비교 분석

[퀀트 전략 일반]

26/2/19 "퀀트 모델링 A to Z - (10) Attention/Transformer의 이해, 1부" – Transformer AI 모델의 이해를 위해 관련된 기초 알고리즘을 설명함
26/2/2 "퀀트 모델링 A to Z - (9) LSTM 모델의 이해" – RNN 계열 핵심 신경망인 LSTM의 작동 원리에 대해 상세히 설명하고 금융에서의 활용법을 소개함
26/1/21 "퀀트 모델링 A to Z - (8) RNN(순환 신경망) AI 투자 모델" – 시계열 및 순차 데이터의 모델링이 뛰어난 RNN 모델의 원리와 활용법을 정리함
26/1/8 "퀀트 모델링 A to Z - (7) CNN AI 투자 모델 (기술적 분석 기반)" - CNN 모델의 전체적인 이해와 사용 방법 등을 설명
26/1/2 "퀀트 모델링 A to Z - (6) MLP(Multi-Layer Perceptron) AI 투자 모델" - 딥러닝을 활용한 퀀트 투자 모델 개발법을 상세히 설명함
25/11/14 "자산배분 전략의 체계" - 자산배분에 대한 기본적인 개념과 접근법을 체계적으로 정리함
25/9/2 "머신 러닝 애널리스트, 파트 1" - 주식 시장의 기업 펀더멘털 분석에서, 머신러닝 및 AI를 활용하는 접근법 제시
25/7/29 " KR-FinBERT의 활용법" – 센티먼트 분석 AI를 손쉽게 사용하는 방법을 정리함
25/5/20 "Corporate Action Book ver 2.0" - 여러 corporate action 이벤트에 대한 메커니즘 설명과 시장 반응의 최근 패턴을 정리함
25/3/20 "MSCI 지수 업데이트 ver 5.0" - MSCI 지수의 최근 이슈와 방법론 상의 변화, 정기 리뷰 로직의 요약 정리 제공
25/1/23 "MSCI 미국 지수 2월 리뷰 편출입 최종 전망" - 24년말부터 MSCI USA 종목 편출입 전망 리포트를 최초 개시함
25/1/14 "TAX 전략 2. 소득세뿐 아니라 건강보험료도 존재한다" - 개인 투자자 관점에서의 건강보험료 및 광의의 세금에 대한 설명 및 전략 자료
24/11/7 "AI와 퀀트 투자" – AI에 의한 세상의 변화와 알고리즘에 대한 이해, 퀀트 투자와의 접목법을 정리함
24/5/27 "KOSPI 지수 장기 수익률 성과 분해" – KOSPI 장기 수익률을 기업이익 증가, 주식수 변동, 밸류에이션으로 분해하여 부진 원인을 정리
24/5/8 "주당 지표 (per share) 기준의 펀더멘털 접근법" – 지수 차원에서의 주당 지표 산출법과 이를 통한 시장 분석, 관련 팩터 전략 등을 정리
24/1/29 "이전 상장 이벤트 및 활용 전략" - 코스피 이전 상장 종목에 대한 주가 변화 결정 요인 상세 분석
23/7/11 "한미 간 자사주 매입의 구조적 차이" – 자사주 매입에 대한 한국과 미국 제도를 비교하고, 시가총액 정의 차이에 따른 문제점을 정리
23/4/17 "NSS (Nelson Siegel Svensson) 모델을 활용한 채권 퀀트 전략, 파트2" - NS 및 NSS 모델을 활용한 실전적인 채권 퀀트 전략 제시
22/8/1 "주식과 채권의 상관성, 파트2 - 주식과 채권 간 상관 관계의 다이내믹스" - 주식/채권 상관관계의 변화를 경기와 물가 요인으로 설명 및 전망
20/10/28 "한국이 MSCI 선진시장에 들어가야 하는 이유" – MSCI 선진시장 승격의 당위성과 승격 시의 변화 등에 대해 총정리
19/11/18 "코스피 단기예측 AI 모델 - 랜덤 포레스트 기법을 활용한 머신러닝 기반 모델" – 랜덤 포레스트 기반의 시장 예측 모델 및 소스 코드 제시
17/3/8 "Volatility Drag - 레버리지 투자 수익과 변동성 손실의 관계" – 변동성 손실 개념과 레버리지 상품의 위험성에 대한 상세 설명
16/1/4 "기계학습(Machine Learning)과 투자전략 - 빅 데이터 기법: Naive Bayes Classifier의 활용" – 머신 러닝에 기반한 퀀트 전략 개발
14/4/24 "CAPE in Everywhere - Cyclically Adjusted P/E의 다용도 활용 분석" – 한국 증시의 CAPE 지표에 대한 본격 분석
11/5/17 "MSCI 종목편입 효과 실증분석 - 종목 편출입에 따른 최적 투자전략" – MSCI 이벤트 활용 투자 전략의 시작

Appendix. Samsung Q.Pack 메일링 자료

글로벌 주식시장 동향부터, FICC, 경제지표, 국내외 종목 및 ETF 정보까지 수록함. 이메일을 통해 정기적으로 엑셀 자료 전송

Index	Region /Country	Current Index (p)	Index change (%) (a=b+c)								Fwd EPS change (%) (b)				Fwd PER change (%) (c)				EPS Growth (% y-y)			PER (x)			
			1D	2D	1W	MTD	1M	3M	YTD	1Y	1M	3M	YTD	1Y	1M	3M	YTD	1Y	Fwd	21E	22E	Fwd	21E	22E	4yAvg
MSCI	AC World	742.26	0.0	0.0	1.3	4.6	2.5	2.7	14.9	28.0	0.2	4.1	24.6	32.9	2.3	-1.3	-7.9	-3.7	12.2	49.3	7.7	**18.5**	19.7	18.3	16.6
Index	World (DM)	3,151.16	0.0	0.1	1.3	4.8	2.5	3.4	17.1	30.2	0.6	4.7	26.0	34.1	1.9	-1.2	-7.1	-2.9	12.4	48.2	7.9	**19.5**	20.9	19.3	17.3
	N. America	4,627.53	-0.2	0.1	1.6	5.8	3.5	4.1	20.8	32.4	1.0	4.7	28.8	35.6	2.5	-0.6	-6.2	-2.3	12.3	48.0	8.4	**21.5**	23.1	21.4	18.7
	Europ																								
	EMU																								
	Pacifi																								
	Emerg																								
	EM A																								
	EM L																								
	EM E																								
	USA																								
	Fran																								
	Gerr																								
	UK																								
	Aust																								
	Japa																								
	Hon																								
	Chin																								
	India																								
	Kore																								
	Taiw																								
	Thai																								
	Indo																								
	Braz																								

Code	Name	연간실적																분기실적						
		매출액(십억)		매출액y-y(%)		영업이익(십억)		영업이익y-y(%)		순이익(십억)		순이익y-y(%)		지배순이익(십억)		지배순이익y-y(%)		매출액(십억)				매출액y-y(%)		
		2021E	2022E	2021E	2022E	2021E	2022E	2021E	2022E	2021E	2022E	2021E	2022E	2021E	2022E	2021E	2022E	1Q21	2Q21	3Q21	4Q21	1Q21	2Q21	3Q21
A005930	삼성전자	276,340	297,907	16.7	7.8	52,686	56,360	46.4	7.0	39,936	43,047	51.2	7.8	38,912	42,428	49.1	9.0	65,389	63,672	73,000	74,295	18.2	20.2	9.0
A000660	SK하이닉스	42,530	49,328	33.3	16.0	12,035	13,823	140.1	14.9	9,128	10,460	91.8	14.6	9,113	10,425	91.7	14.4	8,494	10,322	11,814	11,902	18.0	19.9	45.3
A035420	NAVER	6,784	8,312	27.9	22.5	1,360	1,756	11.9	29.2	15,824	1,825	1,772.6	-88.5	16,582	1,854	1,554.8	-88.8	1,499	1,664	1,727	1,900	-13.5	-12.6	1,000.3
A207940	삼성바이오로직스	1,491	1,690	28.0	13.4	490	531	67.4	8.4	396	435	64.3	9.8	396	435	64.2	9.8	261	412	411	421	25.9	34.0	49.7
A051910	LG화학	44,085	49,876	46.6	13.1	5,849	4,968	225.3	-15.1	4,439	3,476	550.5	-21.7	4,077	3,173	695.2	-22.2	9,650	11,456	11,285	11,733	35.6	74.9	50.3
A035720	카카오	5,987	7,607	44.0	27.1	786	1,147	72.3	46.0	951	1,097	448.6	15.3	902	1,029	479.4	14.1	1,258	1,352	1,644	1,733	44.9	41.9	49.4
A006400	삼성SDI	13,871	16,666	22.8	20.2	1,170	1,577	74.3	34.8	1,194	1,498	89.2	25.5	1,119	1,410	94.6	26.0	2,963	3,334	3,617	3,957	23.6	30.3	17.2
A005380	현대차	116,993	125,659	12.5	7.4	7,136	8,018	198.0	12.4	6,635	7,393	244.7	11.4	6,020	6,795	322.7	12.9	27,391	30,326	28,132	31,054	8.2	38.7	2.0
A068270	셀트리온	2,047	2,419	10.7	18.2	820	963	15.2	17.4	654	762	26.0	16.4	641	746	25.5	16.4	457	432	486	584	22.6	0.7	-11.4
A000270	기아	70,551	75,507	19.2	7.0	5,320	5,854	157.5	10.0	4,733	5,190	218.1	9.7	4,730	5,181	217.9	9.5	16,582	18,339	17,125	18,478	13.8	61.3	4.9
A323410	카카오뱅크					296	521	141.8	75.9	244	399	114.3	64.0	244	399	114.3	64.0	225	254					
A005490	POSCO	72,925	74,829	26.2	2.6	8,884	8,158	269.7	-8.2	6,603	5,877	269.3	-11.0	6,094	5,425	280.4	-11.0	16,069	18,292	20,610	19,280	10.5	33.3	44.5
A012330	현대모비스	41,047	45,462	12.1	10.8	2,228	2,706	21.8	21.4	2,616	3,102	71.3	18.6	2,599	3,081	70.0	18.5	9,816	10,285	9,900	11,004	16.5	36.5	-0.9
A259960	크래프톤	2,128	3,306	27.4	55.4	907	1,531	17.2	68.9	722	1,197	29.8	65.9	722	1,197	29.8	65.9	461	459	460	757	-9.3	21.2	31.4
A028260	삼성물산	32,740	32,926	8.4	0.6	1,407	1,448	64.2	2.9	1,923	1,796	65.7	-6.6	1,730	1,563	67.1	-9.6	7,840	8,548	8,034	8,377	12.6	18.3	2.3
A066570	LG전자	71,956	74,202	13.7	3.1	4,288	4,993	34.2	16.4	2,109	3,675	2.2	74.3	1,821	3,615	-7.5	98.5	18,809	16,117	18,785	17,986	27.7	25.6	11.0
A096770	SK이노베이션	45,612	52,196	33.5	14.4	1,938	2,132	175.5	10.0	901	1,430	142.0	58.8	878	1,401	140.4	59.5	9,240	11,120	12,270	13,120	-17.2	54.5	45.7
A017670	SK텔레콤	19,538	20,303	4.9	3.9	1,526	1,678	13.1	10.0	2,655	2,905	77.0	9.4	2,634	2,834	75.1	7.6	4,780	4,818	4,914	5,036	7.4	4.7	3.9
A105560	KB금융					6,172	6,351	33.7	2.9	4,497	4,688	28.4	4.2	4,463	4,646	29.2	4.1	17,584	12,242	16,473		-8.8	14.2	56.4
A051900	LG생활건강	8,401	9,115	7.1	8.5	1,313	1,463	7.5	11.5	892	1,005	9.7	12.7	874	984	9.6	12.6	2,037	2,021	2,139	2,203	7.4	13.4	3.3
A055550	신한지주					5,993	6,188	21.6	3.3	4,334	4,484	23.9	3.5	4,292	4,414	25.7	2.8	6,895	6,813			-15.3	11.6	
A302440	SK바이오사이언스	976	3,572	332.4	266.2	438	1,578	1,060.7	260.5	355	1,227	980.7	245.2	355	1,227	980.4	245.3	113	145	173	499	397.4	266.8	78.8
A034730	SK	94,268	100,607	15.2	6.7	4,642	5,335	2,922.4	14.9	3,106	3,609	2,964.8	16.2	1,377	1,642	627.2	19.3	22,250	23,406	23,724	25,547	-6.2	24.7	18.9
A091990	셀트리온헬스케어	1,838	2,179	12.9	18.5	331	438	-8.6	32.5	257	336	7.0	30.8	257	336	7.0	30.8	356	433	488	569	-0.2	3.1	5.2
A361610	SK아이이테크놀로	676	1,076	44.1	59.2	189	333	50.7	76.3	148	255	68.1	72.0	149	246	69.3	65.1	139	155	178	227			
A015760	한국전력	59,706	62,797	1.9	5.2	-1,631	-951	-139.9	41.7	-2,165	-1,710	-203.5	21.0	-2,230	-1,755	-212.0	21.3	15,075	13,519	16,039	15,091	-0.1	3.4	2.1
A011200	HMM	12,561	12,447	95.9	-0.9	6,315	5,669	543.9	-10.2	3,837	4,905	2,995.0	27.8	3,866	4,994	3,020.8	29.2	2,428	2,907	3,592	3,489	84.9	111.4	109.0
A003550	LG	8,303	8,612	25.2	3.7	2,797	2,957	64.3	5.7	3,054	2,606	98.3	-14.7	2,958	2,507	101.8	-15.2	2,180	1,924	2,011	2,214	37.5	28.7	2.8
A032830	삼성생명	20,270	20,905	-41.3	3.1	1,182	689	-34.0	-41.7	1,755	1,399	28.1	-20.3	1,653	1,359	30.6	-17.8	10,008	8,195	4,701	4,759	-3.5	7.3	-39.5
A086790	하나금융지주					4,393	4,649	14.5	5.8	3,259	3,416	21.4	4.8	3,200	3,357	21.4	4.9	14,011	6,908	13,195		-32.0	10.8	102.5

Compliance notice

- 본 조사자료는 당사의 저작물로서 모든 저작권은 당사에게 있습니다.
- 본 조사자료는 당사의 동의 없이 어떠한 경우에도 어떠한 형태로든 복제, 배포, 전송, 변형, 대여할 수 없습니다.
- 본 조사자료에 수록된 내용은 당사 리서치센터가 신뢰할 만한 자료 및 정보로부터 얻어진 것이나, 당사는 그 정확성이나 완전성을 보장할 수 없습니다. 따라서 어떠한 경우에도 본 자료는 고객의 주식투자의 결과에 대한 법적 책임소재에 대한 증빙자료로 사용될 수 없습니다.
- 본 자료에는 외부의 부당한 압력이나 간섭 없이 애널리스트의 의견이 정확히 반영되었습니다.

삼성증권

삼성증권주식회사
서울특별시 서초구 서초대로74길 11(삼성전자빌딩)
Tel: 02 2020 8000 / www.samsungpop.com

삼성증권 Family Center: 1588 2323

고객 불편사항 접수: 080 911 0900

Member of
Dow Jones
Sustainability Indices
Powered by the S&P Global CSA