


DNN_03_1. RNN

🕒 생성일	@2022년 6월 10일 오후 1:49
🏷️ 유형	머신러닝/딥러닝
👤 작성자	 동훈 오

핸즈온: 15. RNN과 CNN을 사용해 시퀀스 처리하기

RNN

순환 신경망 (RNN; recurrent neural network) 은 시계열 데이터를 분석해서 주식 가격 같은 것을 예측해 언제 사고팔지 알려 줄 수 있으며, 자동차의 이동 경로를 예측하고 사고를 피하도록 도울 수 있다. 일반적으로 RNN 은 고정 길이 입력이 아닌 임의의 길이를 가진 시퀀스를 다룰 수 있다.

순환 뉴런과 순환 층

순환 신경망은 피드포워드 신경망과 매우 비슷하지만 뒤쪽으로 순환하는 연결이 있다는 점에서 다르다. 아래의 그림은 각 타임 스텝 t 마다 이 순환 뉴런은 물론 x_t 와 이전 타임 스텝의 출력인 $y_{(t-1)}$ 을 입력으로 받는다.

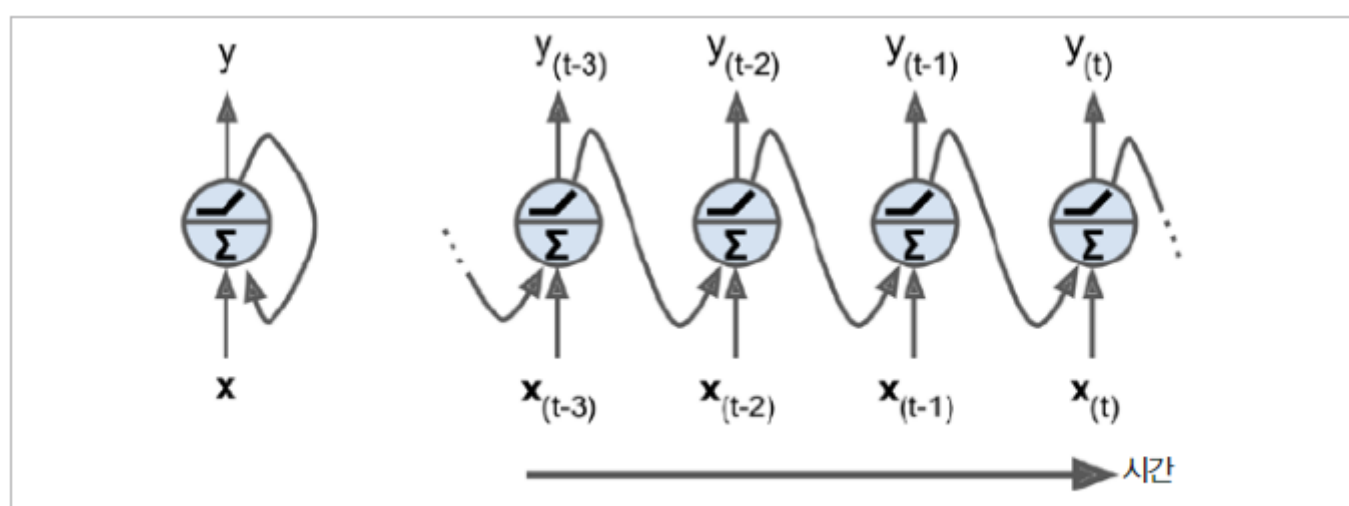


그림 15-1 순환 뉴런(왼쪽)과 타임 스텝으로 펼친 모습(오른쪽)

각 순환 뉴런은 두 개의 가중치를 가진다. 하나는 입력 x_t 를 위한 것이고 다른 하나는 이전 타임 스텝의 출력 $y_{(t-1)}$ 을 위한 것이다. 하나의 순환 뉴런이 아니라 순환 층 전체를 생각하면 각각의 가중치 행렬이 생긴다. 아래의 식은 순환 층 전체의 출력 벡터를 나타낸다.

식 15-1 하나의 샘플에 대한 순환 층의 출력

$$\mathbf{y}_{(t)} = \phi\left(\mathbf{W}_x^T \mathbf{x}_{(t)} + \mathbf{W}_y^T \mathbf{y}_{(t-1)} + \mathbf{b}\right)$$

(ϕ 는 ReLU 와 같은 활성화 함수이다.)

미니배치에 있는 전체 샘플에 대한 순환 뉴런 층의 출력은 다음과 같다.

식 15-2 미니배치에 있는 전체 샘플에 대한 순환 뉴런 층의 출력

$$\begin{aligned} \mathbf{Y}_{(t)} &= \phi(\mathbf{X}_{(t)} \mathbf{W}_x + \mathbf{Y}_{(t-1)} \mathbf{W}_y + \mathbf{b}) \\ &= \phi([\mathbf{X}_{(t)} \mathbf{Y}_{(t-1)}] \mathbf{W} + \mathbf{b}) \quad \text{여기에서} \quad \mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix} \end{aligned}$$

메모리 셀

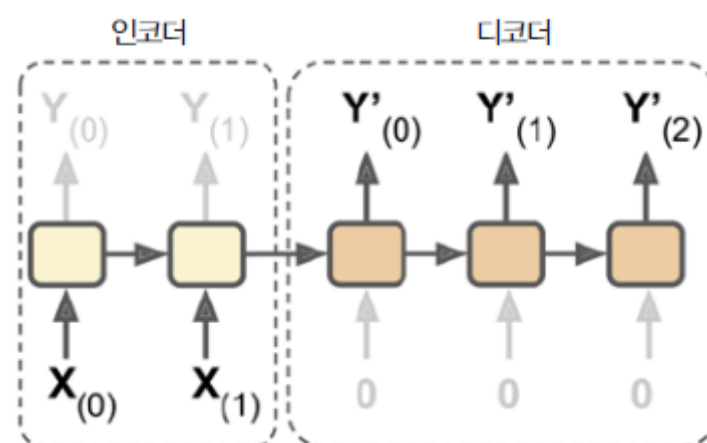
타임 스텝 t 에서 순환 뉴런의 출력은 이전 타임 스텝의 모든 입력에 대한 함수이므로 이를 일종의 메모리 형태라고 말할 수 있다. 타임 스텝에 걸쳐서 어떤 상태를 보존하는 신경망의 구성 요소를 메모리 셀 또는 셀(cell)이라고 한다.

일반적으로 타임 스텝 t 에서의 셀의 상태 h_t 는 그 타임 스텝의 입력과 이전 타임 스텝의 상태에 대한 함수이다. h_t 는 hidden state, 은닉 상태라고 부른다.

입력과 출력 시퀀스

입력 시퀀스를 받아 출력 시퀀스를 만드는 RNN 모델을 Seq2Seq 네트워크라고 한다. 입력 시퀀스를 네트워크에 주입하고, 마지막을 제외한 모든 출력을 무시하는 경우 시퀀스-투-벡터 네트워크 라고 한다. 반대로 각 타임 스텝에서 하나의 입력 벡터를 반복해서 네트워크에 주입하고, 하나의 시퀀스를 출력하는 경우, 벡터-투-시퀀스 네트워크라고 한다.

인코더-디코더 라고 부르는 이중 단계 모델은 seq2seq 네트워크를 구현한다.



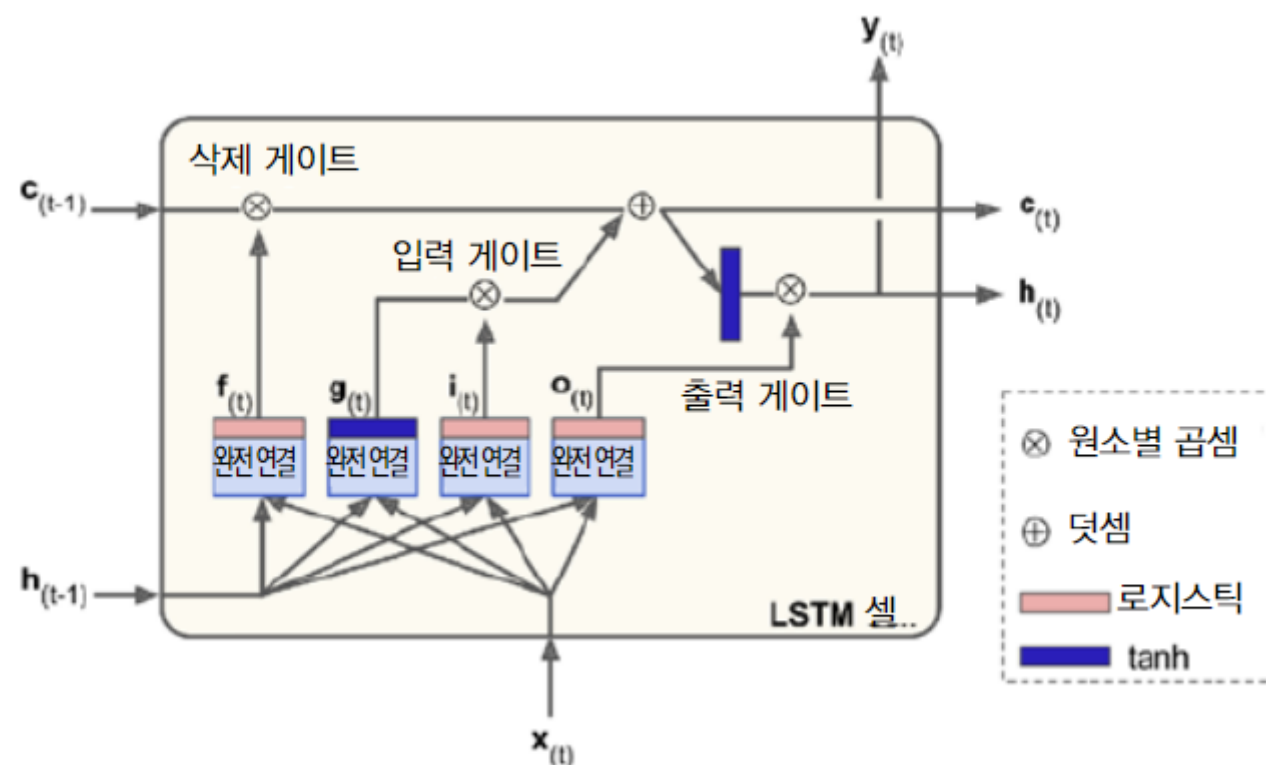
긴 시퀀스 다루기

긴 시퀀스로 RNN을 훈련하려면 많은 타임 스텝에 걸쳐 실행해야 하므로 펼친 RNN 이 매우 깊은 네트워크가 된다. 그래디언트 소실과 폭주 문제를 가지거나 훈련 시간이 오래 걸릴 수 있다. 또한 RNN이 입력의 첫 부분을 조금씩 잊어버리는 문제가 생긴다.

LSTM; long short term memory

위에서 언급된, 긴 시퀀스를 다루는 데 발생하는 현상 중 하나는 시퀀스 데이터 첫 부분의 정보를 점점 잃을 수 있다는 것이다. 장기 메모리를 가진 셀일 구현하여 이를 해결할 수 있다.

LSTM 의 구조를 나타내면 다음 이미지와 같다.



- h_t 는 hidden state : 단기 상태, c_t 는 cell state : 장기 상태 라고 생각하면 된다.

핵심 아이디어는 네트워크가 장기 상태에 저장할 것, 버릴 것, 그리고 읽어들이 것을 학습하는 것이다.

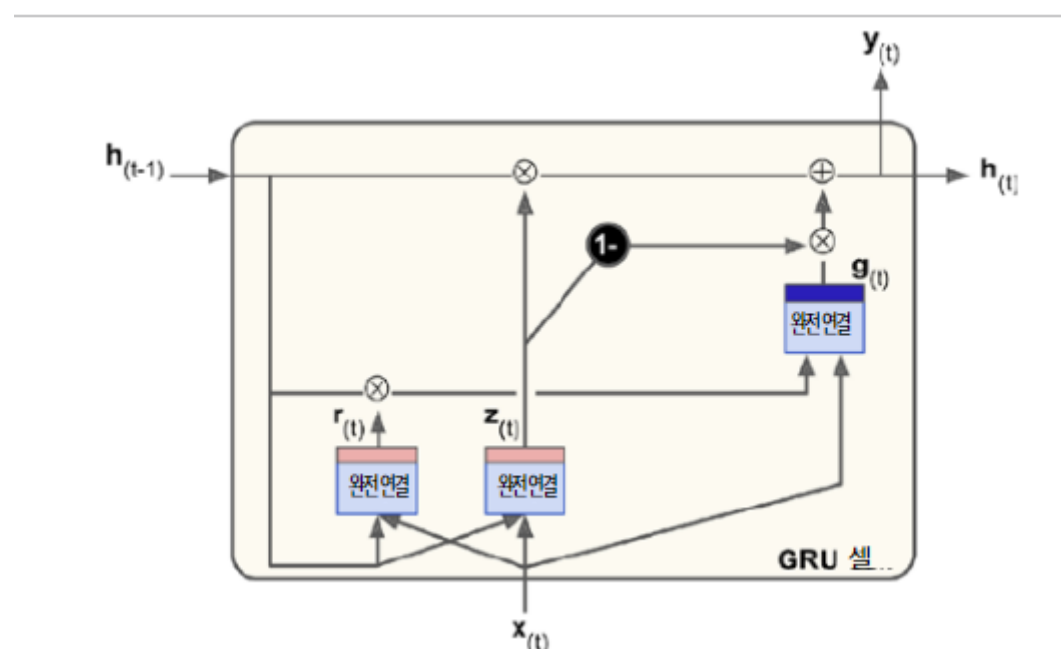
장기 기억 은 네트워크를 왼쪽에서 오른쪽으로 관통하면서, 삭제 게이트를 지나 일부 기억을 잃고, 그런 다음 덧셈 연산으로 새로운 기억 일부를 추가한다.

만들어진 c_t 는 다른 추가 변환 없이 바로 출력으로 보내진다. 그래서 타임 스텝마다 일부 기억이 삭제되고 일부 기억이 추가된다.

또한 덧셈 연산 이후 이 cell state 가 복사되어 tanh 함수로 전달된다. 그런 다음 이 결과는 출력 게이트에 의해 걸러진다. 이는 단기 상태 h_t 를 만든다.

GRU

GRU; gated recurrent unit 은 LSTM 이 간소화된 버전이다. 셀의 구조는 아래 이미지에서 나타난다.



- 두 상태 벡터가 하나의 벡터 h_t 로 합쳐졌다.
- 하나의 게이트 제어기 z_t 가 삭제 게이트와 입력 게이트를 모두 제어한다. 게이트 제어기가 1을 출력하면 삭제 게이트가 열리고, 입력 게이트가 닫힌다. 0을 출력하면 그 반대이다.
- 출력 게이트가 없다. 즉, 전체 상태 벡터가 매 타임 스텝마다 출력된다. 이전 상태의 어느 부분이 주 층 g_t 에 노출될 지 제어하는 새로운 게이트 제어기 r_t 가 있다.

다음의 식은 하나의 샘플에 대해 타임 스텝마다 셀의 상태를 어떻게 계산하는지 요약한다.

식 15-4 GRU 계산

$$\begin{aligned} \mathbf{z}_{(t)} &= \sigma\left(\mathbf{W}_{xz}^T \mathbf{x}_{(t)} + \mathbf{W}_{hz}^T \mathbf{h}_{(t-1)} + \mathbf{b}_z\right) \\ \mathbf{r}_{(t)} &= \sigma\left(\mathbf{W}_{xr}^T \mathbf{x}_{(t)} + \mathbf{W}_{hr}^T \mathbf{h}_{(t-1)} + \mathbf{b}_r\right) \\ \mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}^T \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T (\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}) + \mathbf{b}_g\right) \\ \mathbf{h}_{(t)} &= \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + (1 - \mathbf{z}_{(t)}) \otimes \mathbf{g}_{(t)} \end{aligned}$$

핸즈온: 16. RNN 과 어텐션을 사용한 자연어 처리

IMDB 리뷰 데이터 감성 분석 진행.

데이터로드

```
tf.random.set_seed(42)
(X_train, y_train), (X_test, y_test) = keras.datasets.imdb.load_data()
```

이 데이터셋은 이미 전처리되어 있다. 각 리뷰는 정수 배열로 표현되어 있고 (정수 인코딩된 상태), 각 정수는 하나의 단어를 나타낸다. 구두점을 모두 제거하고 단어는 소문자로 변환한 다음 공백을 기준으로 나누어 빈도에 따라 인덱스를 붙였다. 정수 0,1,2 는 각각 패딩 토큰, SOS 토큰, UNKNOWN 단어를 의미한다.

데이터 전처리

텐서플로 팀의 TF.Text 라이브러리는 wordpiece 를 포함한 다양한 토큰화 전략이 구현되어 있다.

```
# 우선 데이터를 tfds 로 불러온다.
import tensorflow_datasets as tfds
dataset, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)

# 전처리 함수
def preprocess(X_batch, y_batch):
```

```
X_batch = tf.strings.substr(X_batch, 0, 300)
X_batch = tf.strings.regex_replace(X_batch, rb"<br\s*/?>", b" ")
X_batch = tf.strings.regex_replace(X_batch, b"^[a-zA-Z']", b" ")
X_batch = tf.strings.split(X_batch)
return X_batch.to_tensor(default_value=b"<pad>"), y_batch
```

- 리뷰 텍스트를 잘라내어 처음 300 글자만 남기남.
- 정규 표현식을 사용해
 태그 (줄 공백 태그) 를 공백으로 바꾼다. 문자와 작은 따옴표가 아닌 다른 모든 문자를 공백으로 바꾼다.
- 마지막으로 preprocess() 함수는 리뷰를 공백으로 나눈다.

다음으로 vocabulary 를 구축한다. 전체 훈련 세트를 한 번 순회하면서 preprocess() 함수를 적용하고 Counter 로 단어의 등장 횟수를 센다.

```
from collections import Counter
vocabulary = Counter()
for X_batch, y_batch in datasets["train"].batch(32).map(preprocess):
    for review in X_batch:
        vocabulary.update(list(review.numpy()))

vocabulary.most_common()[:3]
>>> [(b'<pad>', 214309), (b'the', 61137), (b'a', 38564)]

# 가장 많이 등장하는 단어 10000개만 남기고 삭제
vocab_size = 10000
truncated_vocabulary = [
    word for word, count in vocabulary.most_common()[:vocab_size]]
```

batch() 는 데이터의 크기를 설정한다.

이제 각 단어를 어휘 사전의 인덱스로 바꾸는 전처리 단계를 수행한다. (정수 인코딩)

```
words = tf.constant(truncated_vocabulary)
word_ids = tf.range(len(truncated_vocabulary), dtype=tf.int64)

# 룩업 테이블 생성
vocab_init = tf.lookup.KeyValueTensorInitializer(words, word_ids)
num_oov_buckets = 1000
table = tf.lookup.StaticVocabularyTable(vocab_init, num_oov_buckets)
```

oov : out of vocabulary 를 의미한다.

최종적으로 리뷰를 배치로 묶고 preprocess() 함수를 사용해 단어를 짧은 시퀀스로 바꾼다. 이후, 단어 인코딩 한다.

```
def encode_words(X_batch, y_batch):
    return table.lookup(X_batch), y_batch

train_set = datasets["train"].batch(32).map(preprocess)
train_set = train_set.map(encode_words).prefetch(1)
```

prefetch : 학습중일때, 데이터 로드시간을 줄이기 위해 미리 메모리에 적재시킨다. 이때, 괄호안의 숫자는 얼마만큼 적재시킬지에 대한 숫자이다.

모델 작성

```

embed_size = 128
model = keras.models.Sequential([
    keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size,
                           input_shape=[None]),
    keras.layers.GRU(128, return_sequences=True),
    keras.layers.GRU(128),
    keras.layers.Dense(1, activation='sigmoid')
])

model.compile(optimizer='adam', loss='binary_crossentropy',
              metrics=['accuracy'])
history = model.fit(train_set, epochs=5)

```

첫 번째 층은 단어 ID 를 임베딩으로 변환하는 Embedding 층이다. 임베딩 행렬은 단어ID 당 (vocab_size + num_oov_buckets) 하나의 행과 임베딩 차원당 하나의 열을 가진다. 모델의 입력은

[배치 크기, 타임 스텝 수] 크기를 가진 2D 텐서이지만, Embedding 층의 출력은 [배치 크기, 타임 스텝 수, 임베딩 크기] 크기를 가진 3D 텐서가 된다.

모델의 나머지 부분은 GRU 층 두 개로 구성되고 두 번째 층은 마지막 타임스텝의 출력만 반환된다. 출력층은 시그모이드 활성화 함수를 사용하는 하나의 뉴런이다. 리뷰가 영화에 대한 긍정적인 감정을 표현하는지에 대한 추정 확률을 출력한다.

마스킹(masking)

원래 데이터 그대로 모델이 패딩 토큰을 무시하도록 학습되어야 한다. 그런데 이미 무시할 토큰을 알고 있다. 패딩 토큰을 무시하도록 모델에게 알려주어 실제 의미가 있는 데이터에 집중할 수 있게 만드는 것이 당연하다.

Embedding 층을 만들 때 mask_zero=True 매개변수를 추가하면 된다. 이렇게 하면 이어지는 모든 층에서 패딩 토큰(=0)을 무시하게 된다.

다음 모델은 이전 모델과 동일하지만 함수형 API를 사용하여 직접 마스킹을 처리한다.

```

K = keras.backend
inputs = keras.layers.Input(shape=[None])
mask = keras.layers.Lambda(lambda inputs: K.not_equal(inputs, 0))(inputs)
z = keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size)(inputs)
z = keras.layers.GRU(128, return_sequences=True)(z, mask=mask)
z = keras.layers.GRU(128)(z, mask=mask)
outputs = keras.layers.Dense(1, activation='sigmoid')(z)
model = keras.Model(inputs=[inputs], outputs=[outputs])

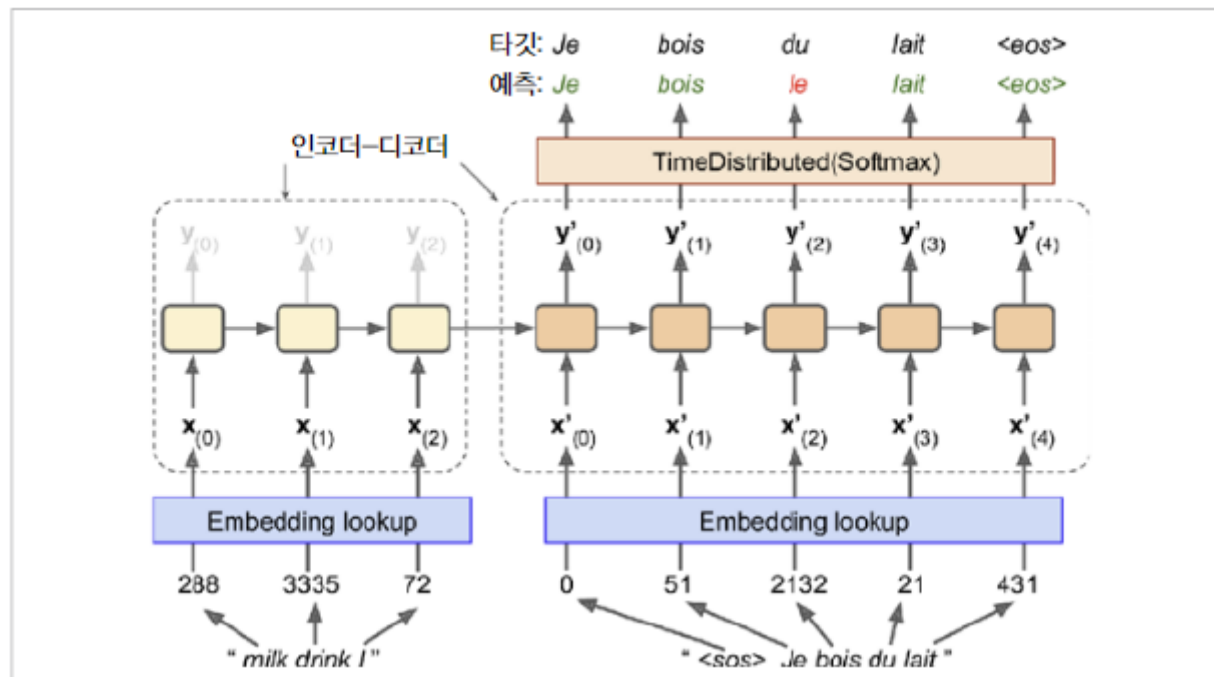
```

신경망 기계 번역을 위한 인코더-디코더 네트워크

영어 문장을 프랑스어로 번역하는 간단한 신경망 기계 번역 모델을 살펴본다. 간략하게 정리하면 영어 문장을 인코더로 주입하면 디코더는 프랑스어 번역을 출력한다. 프랑스어 번역은 한 스텝 뒤쳐져서 디코더의 입력으로도 사용된다.

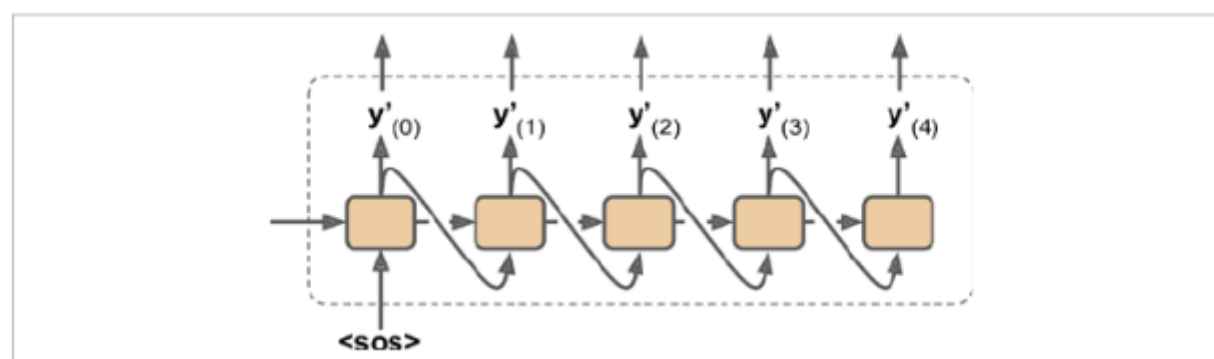
맨 처음 단어는 SOS 토큰으로 시작한다. 디코더는 문장의 끝에 EOS 토큰이 있을 것으로 기대한다.

각 단어는 초기에 1차원으로 표현되어 있다. 그다음 임베딩 층이 단어 임베딩을 반환한다. 이 단어 임베딩이 인코더와 디코더로 주입된다.



각 단계마다 디코더는 출력 어휘 사전에 있는 단어에 대한 점수를 출력한다. 그다음 소프트맥스 층이 이 점수를 확률로 바꾼다. 이는 일반적인 분류 작업과 매우 비슷하다.

test 시에는 디코더에 주입할 타깃 문장이 없다. 대신 그냥 이전 스텝에서 디코더가 출력한 단어를 주입한다.



- 인코더와 디코더에 input 으로 사용되는 문장의 길이는 다를 수 밖에 없다. 일반적인 텐서는 크기가 고정되어 있기에 동일한 길이의 문장만 담을 수 있고, 마스킹을 사용한다면 문장 길이를 동일하게 편집할 수 있다.
- 감성 분석과 달리 기계 번역에서는 문장 길이를 잘라낼 수 없다. 대신 문장을 비슷한 길이의 버킷으로 그룹화한다. 버킷에 담긴 문장이 모두 동일한 길이가 되도록 패딩을 추가한다.
- EOS 토큰 이후 출력은 모두 무시한다. 이 토큰들은 마스킹 처리되어 손실에 영향을 미치지 않는다.

tensorflow addon 프로젝트에서 제공하는 여러가지 seq2seq 도구를 활용하면 인코더-디코더 구조를 손쉽게 만들 수 있다.

```

vocab_size = 100
embed_size = 10

import tensorflow_addons as tfa

encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)

embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)

encoder = keras.layers.LSTM(512, return_state=True)
# 인코더의 결과로 리턴되는 값들이 무엇인지, 순서가 어떤지 중요하다!
encoder_outputs, state_h, state_c = encoder(encoder_embeddings)
encoder_state = [state_h, state_c]

sampler = tfa.seq2seq.sampler.TrainingSampler()

```

```

decoder_cell = keras.layers.LSTMCell(512)
output_layer = keras.layers.Dense(vocab_size)
decoder = tf.nn.seq2seq.basic_decoder.BasicDecoder(decoder_cell, sampler,
                                                    output_layer=output_layer)

final_outputs, final_state, final_sequence_lengths = decoder(
    decoder_embeddings, initial_state=encoder_state,
    sequence_length=sequence_lengths)

Y_proba = tf.nn.softmax(final_outputs.rnn_output)

model = keras.Model(inputs=[encoder_inputs, decoder_inputs, sequence_lengths],
                    outputs=[Y_proba])

```

- LSTM 층을 만들 때 최종 은닉 상태를 디코더로 보내기 위해 return_state=True 로 지정했다. LSTM 셀을 사용하기 때문에 은닉 상태 두 개를 반환한다.
 - TrainingSampler 는 텐서플로 애드온에 포함되어 있는 여러 샘플러 중 하나이다. 이 샘플러는 각 스텝에서 디코더에게 이전 스텝의 출력이 무엇인지 알려준다.
- 추론 시에는 실제로 출력되는 토큰의 임베딩이 된다. 훈련 시에는 이전 타깃 토큰의 임베딩이 되어야 한다.

양방향 RNN

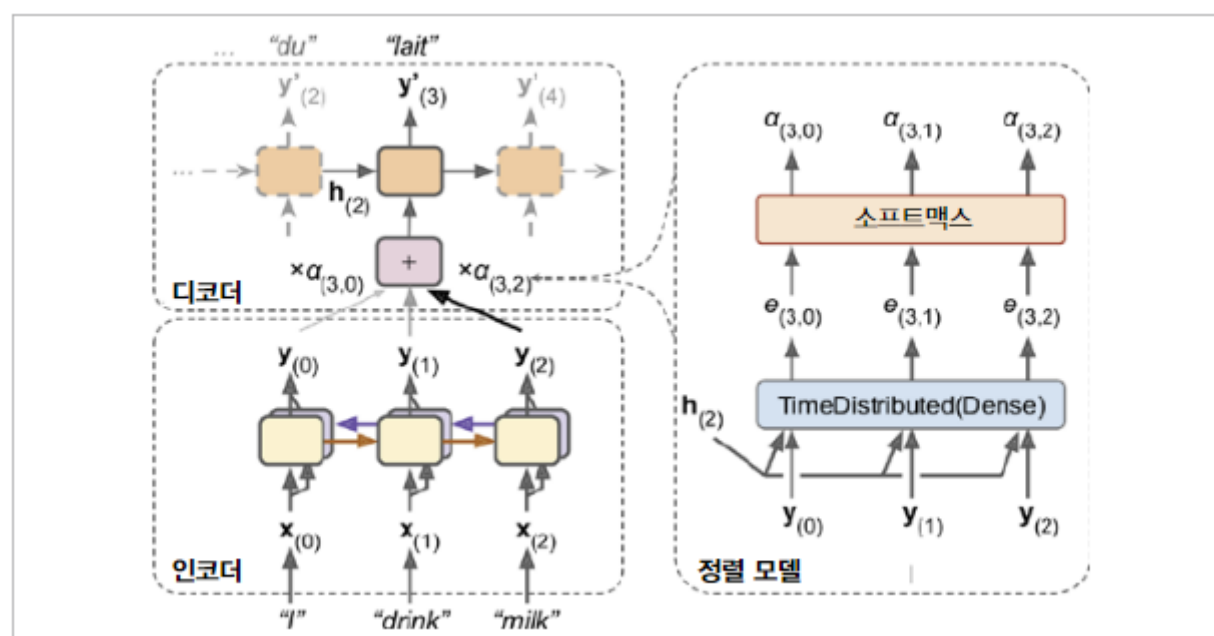
양방향 순환 층은 동일한 입력에 대해 두 개의 순환 층을 실행한다. 하나는 왼쪽에서 오른쪽으로 단어를 읽고 다른 하나는 오른쪽에서 왼쪽으로 읽는다. 그다음 일반적으로 타임 스텝마다 이 두 출력을 연결한다.

beam search

모델이 실수를 고칠 수 있게 하는 방법 중 하나가 빔 검색이다. k개의 가능성 있는 문장의 리스트를 유지하고 디코더 단계마다 이 문장의 단어를 하나씩 생성하여 가능성 있는 k개의 문장을 만든다. 파라미터 k 를 beam width 라고 부른다.

Attention

Bahdanau 의 논문에서 각 타임 스텝에서 적절한 단어에 디코더가 초점을 맞추도록 하는 기술이 소개되었다. 입력 단어에서 번역까지 경로가 훨씬 짧아지므로 RNN 의 단기 기억의 제한성에 훨씬 적은 영향을 받게 된다.



각 타임 스텝에서 디코더의 메모리 셀은 모든 인코더 출력의 가중치 합을 계산한다. 이 단계에서 주의 집중할 단어를 결정한다. $a_{(t,i)}$ 가중치는 t 번째 디코더 타임 스텝에서 i 번째 인코더 출력의 가중치이다. 타임 스텝마다 메모리 셀이 앞서 언급한 입력과 이전 타임 스텝의 은닉 상태를 받는다.

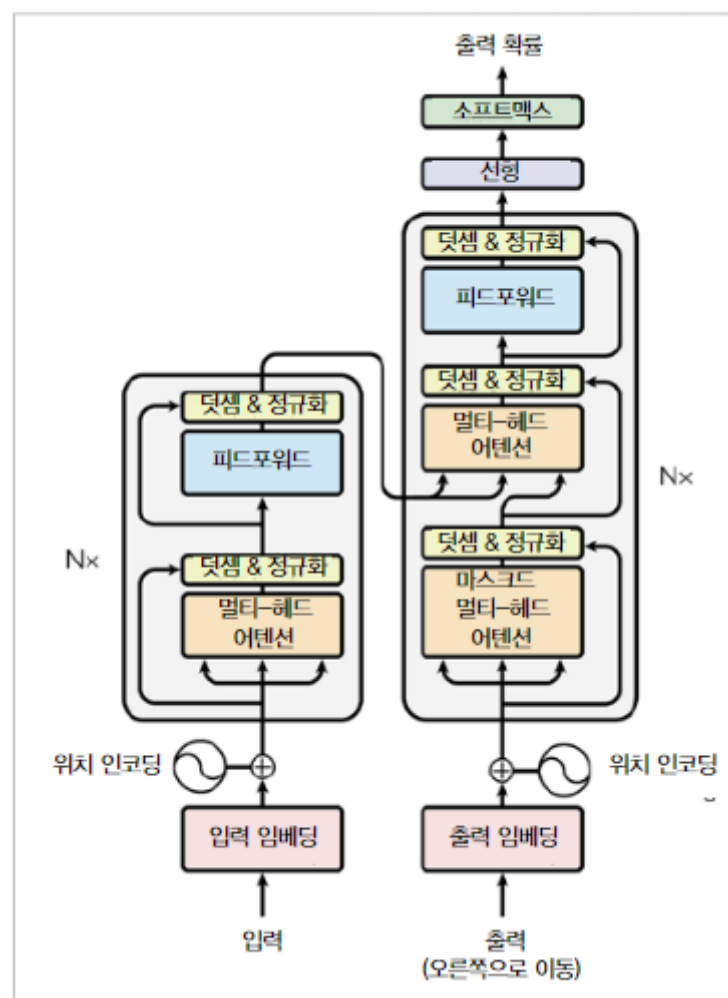
$a_{(t,i)}$ 가중치는 attention layer 이라고 부르는 작은 신경망에서 생성된다. TimeDistributed 클래스를 적용한 Dense 층으로 시작한다. 이 층은 하나의 뉴런으로 구성되고 인코더의 모든 출력을 입력으로 받아 디코더의 이전 은닉 상태를 연결한다. 이 층은 각 인코더 출력에 대한 점수를 출력한다. 이 점수는 각 출력이 디코더의 은닉 상태와 얼마나 잘 맞는지를 측정한다. 마지막으로 소프트맥스 층을 통과해 최종 가중치를 얻는다. (주어진 디코더 타임 스텝에 대한 모든 가중치 합은 1이다.) 이런 종류의 어텐션을 Bahdanau attention 이라고 한다.

또다른 어텐션은 dot product attention이 있다. 이 어텐션의 목적은 인코더의 출력 하나와 디코더의 이전 은닉 상태 사이의 유사도를 측정하는 것이기에 간단히 두 벡터 사이의 dot product 를 사용한다. (이렇게 하려면 두 벡터는 동일한 차원을 가져야 한다.)

dot product 의 결과는 하나의 점수인데, 여러 개의 점수가 소프트맥스를 통과해 최종 가중치를 만든다.

Transformer: Attention is all you need.

2017년 구글 연구팀이 제안한 트랜스포머 구조는, 순환 층이나 합성곱 층을 전혀 사용하지 않고 어텐션 메커니즘(+ 임베딩 층, 밀집층, 정규화 층) 을 사용해 기계 번역 문제에서 성능을 크게 향상시켰다. 다음 이미지는 트랜스포머의 구조를 나타낸다.



- 왼쪽 부분은 인코더이다. 정수 인코딩 된 시퀀스를 입력으로 받는다. 입력의 크기는 [배치 크기, 문장의 최대 길이] 이다. 논문에 따르면 인코더는 각 단어를 512 차원의 표현으로 인코딩 한다.

⇒ 인코더 출력 크기 : [배치 크기, 문장 최대 길이, 512차원]

인코더의 뒷부분은 N 번 반복되어 쌓아 올린다.

- 오른쪽 부분은 디코더이다.

- 훈련 과정 :

타깃 문장을 입력으로 받는다. 시작 부분에 sos 토큰이 추가되어 있어 타임스텝이 하나씩 밀려나 있게 된다.

인코더의 출력을 받고, 디코더의 윗부분도 N 번 반복되어 쌓아 올린다. 인코더의 최종 출력이 N 번의 디코더에 모두 주입된다. 앞선 RNN 모델과 동일하게, 타임스텝마다 디코더는 가능한 다음 단어에 대한 확률을 출력한다.

⇒ 디코더 출력크기 : [배치크기, 문장 최대길이, vocab길이]

- 추론 과정:

추론 시에는 디코더에 타깃을 주입할 수 없다. 이전 타임 스텝에서 출력된 단어를 주입한다.

- 구성요소

- 임베딩 층 두개, 스킵 연결 5* N 개, 그 뒤를 따르는 정규화 층, 밀집 층 두 개로 구성된 피드포워드 모듈이 2*N 개 있다. 출력층은 소프트맥스 활성화 함수를 사용하는 밀집 층을 사용한다.

- 각각의 층은 타임 스텝에 독립적이다. (=time distributed) 따라서 각 단어는 다른 모든 단어에 대해 독립적으로 처리된다. 한 단어씩 보면서 문장을 번역하는데 어려움이 있으므로 특별한 구성 요소가 추가된다.

- 인코더-multi head attention

멀티-헤드 어텐션 층은 관련이 많은 단어에 더 많은 주의를 기울이면서 각 단어와 동일한 문장에 있는 다른 단어의 관계를 인코딩한다. 이러한 어텐션 메커니즘을 self-attention 이라고 한다.

- 디코더-masked multi head attention

동일한 작업 수행. 각 단어는 이전에 등장한 단어에만 주의를 기울일 수 있다.

- 디코더-multi head attention

디코더가 입력 문장에 있는 단어에 주의를 기울이는 곳이다.

- 위치 인코딩 (positional encoding)

문장에 있는 단어의 위치를 타나내는 단순한 밀집 벡터이다. n 번째 위치 인코딩이 각 문장에 있는 n 번째 단어의 단어 임베딩에 더해진다. 이를 통해 모델이 각 단어의 위치를 알 수 있게 된다. 이는 멀티 헤드 어텐션 층이 단어 사이 관계만 보고 단어의 순서나 위치를 고려하지 않기 때문에 필요하다.

위치 인코딩

위치 인코딩은 단어의 위치를 인코딩한 밀집 벡터임을 주의한다.

논문에서 저자들은 여러 가지 주기의 사인과 코사인 함수로 정의한 고정된 위치 인코딩을 선호했다. 위치 인코딩 행렬 P 는 다음과 같이 정의된다.

$$\begin{aligned}P_{p, 2i} &= \sin(p / 10000^{2i/d}) \\P_{p, 2i+1} &= \cos(p / 10000^{2i/d})\end{aligned}$$

$P_{p,i}$ 는 문장에서 p 번째 위치에 있는 단어를 위한 인코딩의 i 번째 원소이다.

위치마다 고유한 위치 인코딩이 만들어지기 때문에 위치 인코딩을 단어 임베딩에 더하면 모델이 문장에 있는 단어의 절대 위치를 알 수 있다. 또한 진동함수 선택에 따라 모델이 상대적인 위치도 학습할 수 있다.

PositionalEmbedding 층은 다음과 같이 만들 수 있다. 생성자에서 위치 인코딩 행렬을 미리 계산한다. (문장의 최대 길이 max_steps 와 각 단어를 표현할 차원수 max_dims 를 알아야 한다.) 그 다음 call() 메서드에서 이 인코딩 행렬을 입력의 크기로 잘라 입력에 더한다. 위치 인코딩 행렬을 만들 때 크기가 1인 첫 번째 차원을 추가했으므로 브로드캐스팅 규칙에 의해 이 행렬이 입력의 모든 문장에 더해질 수 있다.

```
class PositionalEncoding(keras.layers.Layer):
    def __init__(self, max_steps, max_dims, dtype=tf.float32, **kwargs):
        super().__init__(dtype=dtype, **kwargs)

        if max_dims % 2 == 1:
            max_dims += 1 # 단어 차원 수는 짝수여야 한다.
        p, i = np.meshgrid(np.arange(max_steps), np.arange(max_dims // 2))
        pos_emb = np.empty((1, max_steps, max_dims)) # 크기가 1인 차원 추가
        # P_{p,2i}
        pos_emb[0, :, ::2] = np.sin(p/10000**(2*i / max_dims)).T
        # P_{p,2i+1}
        pos_emb[0, :, 1::2] = np.cos(p/10000**(2*i / max_dims)).T

        self.positional_embedding = tf.constant(pos_emb.astype(self.dtype))

    def call(self, inputs):
        shape = tf.shape(inputs)
        return inputs + self.positional_embedding[:, :shape[-2], :shape[-1]]
```

- np.meshgrid : numpydml meshgrid 함수는 1차원 좌표 배열 (x1, x2, ... , xn)에서 N 차원 직사각형 격자를 만드는 함수이다.

scaled dot product attention

멀티 헤드 어텐션 층의 작동 방식을 이해하려면 먼저 기본이 되는 scaled dot product attention 을 이해해야 한다.

ex.

인코더가 'They played chess' 와 같은 입력 문장을 분석하여 단어 They 는 주어, 단어 played 는 동사라는 것을 이해하려 한다. 인코더는 이 정보를 단어의 표현에 인코딩한다. 디코더가 이미 주어를 번역했고 다음에 동사를 번역해야 한다고 가정.

→ 입력 문장에서 동사를 추출해야 한다.

→ 이것은 딕셔너리 룩업과 유사하다. 인코더가 {주어: 'They', 동사: 'played', ...} 와 같은 딕셔너리를 만들고 디코더가 '동사' key 에 해당하는 value 를 찾는 것과 같다.

모델이 키를 표현하기 위한 구분되는 토큰을 갖지 않는다. 다만, 모델은 훈련하는 동안 학습된 벡터 표현으로 이 개념을 갖는다.

따라서 룩업에 사용할 키(=query) 는 딕셔너리에 있는 키와 완벽하게 매칭되지 않는다. 한 가지 방법은 쿼리와 딕셔너리에 있는 각 키 사이의 유사도를 계산하는 것이다. 그런 다음 소프트맥스 함수를 사용해 유사도 점수를 합해서 1이 되는 가중치로 바꾼다. 동사를 표현하는 키가 쿼리와 가장 비슷하다면 1에 가까운 큰 값으로 나올 것이다.

그 다음 모델이 키에 해당하는 값의 가중치 합을 계산할 수 있다. 따라서 동사의 키 가중치가 1에 가까우면 이 가중치 합은 단어 played 의 표현에 매우 가깝게 될 것이다.

트랜스포머에서 사용하는 유사도 측정 방법은 dot product 이다. 다음은 벡터 형식으로 이 공식을 나타내었다.

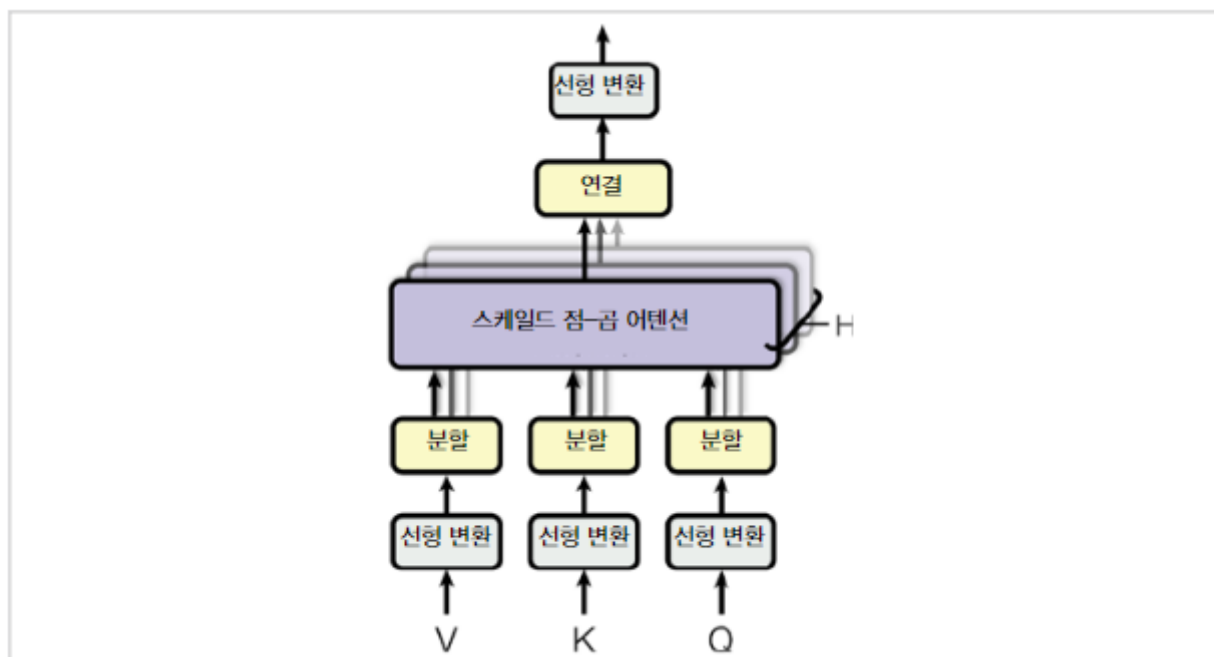
$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{\text{keys}}}}\right)\mathbf{V}$$

- d_{keys} 는 쿼리와 키의 차원 개수이다.

디코더의 위쪽 어텐션 층에서 K, V 는 단순히 인코더가 생성한 단어 인코딩의 리스트이다. Q 는 디코더가 생성한 단어 인코딩의 리스트이다.

keras.layers.Attention 층은 scaled dot product attention 을 구현한다. 이 층의 입력은 첫 번째 차원에 배치 차원이 추가로 있는 것을 제외하면 Q, K, V 와 같다.

multi-head attention



multi head attention 은 scaled dot product attention layer 의 묶음이다. 각 층은 V, K, Q 의 선형 변환이 선행된다. (활성화 함수가 없는, 타임 스텝에 독립적인 Dense 층)

출력은 단순히 모두 연결되어 마지막 선형 변환(활성화 함수 없는 Dense layer)을 통과한다.

하나의 scaled dot product attention 층 만을 사용한다면 한번에 번역에 필요한 특징들을 모두 취리할 수밖에 없다. 멀티 헤드 어텐션을 사용한다면 모델이 단어 표현을 여러 부분 공간으로 다양하게 투영할 수 있다. 이 부분 공간은 단어의 일부 특징에 주목한다. 선형 층의 하나가 단어 표현을 이 단어가 동사라는 정보만 남는 하나의 부분 공간으로 투영한다. 또 다른 선형 층은 과거형이라는 사실만 추출하는 식이다. scaled dot product attention 층이 룩업 단계를 구현하고 마지막의 모든 결과를 연결하여 원본 공간으로 다시 투영한다.