RNN과 어텐션을 사용한 자연어 처리

▼ 16.1 Char-RNN을 사용해 셰익스피어 문체 생성하기

```
PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:
Come, sir, I will make did behold your worship.

VIOLA:
I'll drink it.
```

▼ 1. 훈련 데이터셋 만들기

```
shakespeare_url = "https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt"
filepath = keras.utils.get_file("shakespeare.txt", shakespeare_url)
with open(filepath) as f:
    shakespeare_text = f.read()
```

```
Downloading data from https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt
1122304/1115394 [==============================] - 0s 0us/step
1130496/1115394 [==============================] - 0s 0us/step
```

```
tokenizer = keras.preprocessing.text.Tokenizer(char_level=True)
tokenizer.fit_on_texts(shakespeare_text)
```

```
tokenizer.texts_to_sequences(["First"])
```

```
[[20, 6, 9, 8, 3]]
```

```
tokenizer.sequences_to_texts([[20, 6, 9, 8, 3]])
```

```
['f i r s t']
```

```
max_id = len(tokenizer.word_index) # 고유한 문자 개수
dataset_size = tokenizer.document_count # 전체 문자 개수
```

```
print(max_id, dataset_size)
```

```
39 1115394
```

- 정수 인코딩(케라스의 Tokenizer 클래스 사용)
- Tokenization : Tokenization이란 Text를 여러개의 Token으로 나누는 것을 말합니다.
- 텍스트에서 사용되는 모든 글자를 찾아 각기 다른 글자 ID에 매핑 → 1부터 시작해 고유한 글자 개수까지 만들어진다. (char_level = True ⇒ 글자 수준 인코딩)
- 글자 ID 인코딩 or 디코딩 가능(클래스 자체적으로 텍스트를 소문자로 바꿈)
- 고유 글자 개수와 전체 글자 개수 확인 가능

```
[encoded] = np.array(tokenizer.texts_to_sequences([shakespeare_text])) - 1
```

- 전체 텍스트 인코딩(1부터 시작했으므로 -1 함)

▼ 2. 순차 데이터셋을 나누는 방법

```
train_size = dataset_size * 90 // 100
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
```

- 훈련, 검증, 테스트 세트가 중복되지 않도록 나눠야한다.
- 텍스트의 처음 90%를 훈련 세트로
- 인코딩 하여 한번에 한글자씩 slices로 만들고 tf.data.Dataset 객체를 만든다.

▼ 3. 순차 데이터를 윈도 여러 개로 자르기

- 훈련세트는 백만개 이상의 글자로 이루어진 시퀀스 하나 → 신경망 직접 훈련 불가
- 데이터셋의 window() 메서드를 사용 긴 시퀀스 → 작은 많은 텍스트 윈도

```
n_steps = 100
window_length = n_steps + 1 # 타깃 = 한 글자 앞선 입력
dataset = dataset.window(window_length, shift=1, drop_remainder=True)
```

- 첫번째 윈도는 0~100번째 글자를, 두번째 윈도는 1~101번째 글자를 포함
- drop_remainder =True 모든 윈도가 동일하게 101개의 글자 포함(패딩 없이 배치데이터 만들기)
- 패딩 : 병렬 연산을 위해서 여러 문장의 길이를 임의로 동일하게 맞춰주는 작업
- 중첩 데이터셋: 하나의 데이터셋으로 표현되는 윈도를 포함 데이터셋(리스트의 리스트)

```
dataset = dataset.flat_map(lambda window: window.batch(window_length))
```

- 플랫 데이터셋: 중첩 데이터셋을 평평하게
- batch(window_length) → 텐서 하나를 담은 데이터셋 얻기

```
batch_size = 32
dataset = dataset.shuffle(10000).batch(batch_size)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
```

- 윈도를 섞은 뒤(독립 분포)
- 입력(처음 100개 글자)과 타깃(마지막 100개 글자) 분리

```
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
```

- 범주형 특성 + 고유한 글자 수가 적음(39개) ⇒ 원-핫 벡터 글자 인코딩

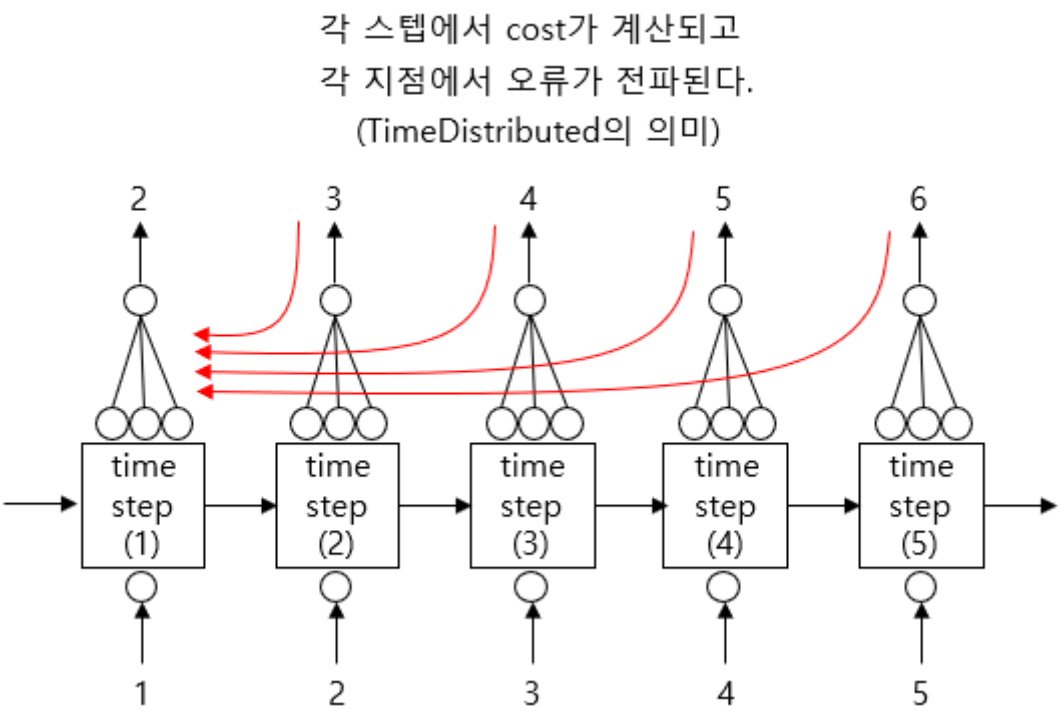
```
dataset = dataset.prefetch(1)
```

- 프리페칭 = 앞으로 연산에 필요한 data 들을 미리 가져 옴

▼ 4. Char-RNN 모델 만들고 훈련하기

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, input_shape=[None, max_id],
                     #dropout=0.2, recurrent_dropout=0.2),
                     dropout=0.2),
    keras.layers.GRU(128, return_sequences=True,
                     #dropout=0.2, recurrent_dropout=0.2),
                     dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                    activation="softmax"))
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
history = model.fit(dataset, epochs=10)
```

- 유닛 128개를 가진 GPU층 2개와 입력과 은닉상태에 20% 드롭아웃(현재는 GPU 사용을 위해 주석처리 상태)
- 출력층은 Timedistributed 클래스를 적용한 Dense 층
- Timedistributed



- 타임 스텝이서 출력 확률의 합 1 → softmax 함수 적용

- sparse_categorical_crossentropy : 다중 분류 손실함수 **one-hot encoding 클래스**
  출력값이 one-hot encoding 된 결과로 나오고 실측 결과와의 비교시에도 실측 결과는 one-hot encoding 형태로 구성된다.
- Optimizer : 학습 데이터(Train data)셋을 이용하여 모델을 학습 할 때 데이터의 실제 결과와 모델이 예측한 결과를 기반으로 잘 줄일 수 있게 만들어주는 역할 (Adam)

▼ 5. Char-RNN 모델 사용하기

```
def preprocess(texts):
    X = np.array(tokenizer.texts_to_sequences(texts)) - 1
    return tf.one_hot(X, max_id)
```

- 모델에 새로운 텍스트를 입력하려면 전처리 과정을 거쳐야 함

```
X_new = preprocess(["How are yo"])
#Y_pred = model.predict_classes(X_new)
Y_pred = np.argmax(model(X_new), axis=-1)
tokenizer.sequences_to_texts(Y_pred + 1)[0][-1] # 1st sentence, last char
```

```
'u'
```

- 텍스트의 다음 글자를 예측

▼ 6. 가짜 셰익스피어 텍스트를 생성하기

- 위의 방식은 글자를 텍스트 끝에 추가하고 늘어난 텍스트를 모델에 전달하여 다음 글자를 예측 → 같은 단어가 계속 반복
- tf.tandom.categorical() 함수를 사용 모델이 추정한 확률을 기반으로 다음 글자를 무작위 선택
- temperature ⇒ 비율 조정
- 온도가 0에 가까움 → 높은 확률을 가진 글자 선택
  온도가 매우 높음 → 모든 글자가 동일한 확률을 갖게 됨
- 다음 글자를 얻는 함수

```
def next_char(text, temperature=1):
    X_new = preprocess([text])
    y_proba = model(X_new)[0, -1:, :]
    rescaled_logits = tf.math.log(y_proba) / temperature
    char_id = tf.random.categorical(rescaled_logits, num_samples=1) + 1
    return tokenizer.sequences_to_texts(char_id.numpy())[0]
```

- 텍스트를 추가하는 함수

```
def complete_text(text, n_chars=50, temperature=1):
    for _ in range(n_chars):
        text += next_char(text, temperature)
    return text
```

- 텍스트 생성

```
print(complete_text("t", temperature=0.2))
```

```
the maid in padua for my father is a stood
and so m
```

```
print(complete_text("t", temperature=1))
```

```
toke on advised in sobel countryman,
and signior gr
```

```
print(complete_text("t", temperature=2))
```

```
tpeniomently!
well maze: yet 'pale deficuruli-faeem
```

- 1에 가장 가까운 온도에서 가장 잘 작동

▼ 7. 상태가 있는 RNN

- **상태가 없는 RNN :** 훈련 반복마다 모델의 은닉 상태를 0으로 초기화, 타임 스텝마다 이 상태를 업데이트, 마지막 타임 스텝 후 버림
- **상태가 있는 RNN :** RNN이 훈련 배치를 처리한 후에 마지막 상태를 다음 훈련 배치에 초기 상태로 사용, 모델이 장기간 패턴을 학습

```
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
dataset = dataset.window(window_length, shift=n_steps, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(window_length))
dataset = dataset.batch(1)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
dataset = dataset.prefetch(1)
```

- 순차적이고 겹치지 않는 입력 시퀀스를 만들어야 함
- window() 메서드에서 shift=n_steps 사용
- shuffle() 메서드 호출 X
- batch(32)라고 호출하면 연속적인 윈도가 같은 배치에 들어가기 때문에 윈도가 끝난 시점부터 배치가 계속되지 않는다. → 하나의 윈도를 갖는 배치를 만듦

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                     #dropout=0.2, recurrent_dropout=0.2,
                     dropout=0.2,
                     batch_input_shape=[batch_size, None, max_id]),
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                     #dropout=0.2, recurrent_dropout=0.2),
                     dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                    activation="softmax"))
])
```

1. stateful = True
2. 배치 크기를 알아야함(입력 시퀀스의 상태 보존) → batch_input_shape 매개변수 지정

```
class ResetStatesCallback(keras.callbacks.Callback):
    def on_epoch_begin(self, epoch, logs):
        self.model.reset_states()
```

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
history = model.fit(dataset, epochs=50,
                    callbacks=[ResetStatesCallback()])
```

```
Epoch 1/50
313/313 [==============================] - 6s 12ms/step - loss: 2.6200
Epoch 2/50
313/313 [==============================] - 4s 12ms/step - loss: 2.2410
Epoch 3/50
313/313 [==============================] - 4s 12ms/step - loss: 2.1105
Epoch 4/50
```

- 에포크 끝마다 텍스트를 다시 시작하기 전에 상태를 재설정해야 한다. (ResetStatesCallback)
- 모델을 컴파일하고 훈련

▼ 16.2 감성 분석

▼ 감성 분석

- 데이터셋 : IMDb 리뷰 데이터셋

```
(X_train, y_train), (X_test, y_test) = keras.datasets.imdb.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz
17465344/17464789 [==============================] - 0s 0us/step
17473536/17464789 [==============================] - 0s 0us/step
```

```
X_train[0][:10]
```

```
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
```

- 이미 정수 배열로 전처리 된 데이터 셋
- 디코딩

```
word_index = keras.datasets.imdb.get_word_index()
id_to_word = {id_ + 3: word for word, id_ in word_index.items()}
for id_, token in enumerate(("<pad>", "<sos>", "<unk>")):
    id_to_word[id_] = token
" ".join([id_to_word[id_] for id_ in X_train[0][:10]])
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb_word_index.json
1646592/1641221 [==============================] - 0s 0us/step
1654784/1641221 [==============================] - 0s 0us/step
'<sos> this film was just brilliant casting location scenery story'
```

- 텐서플로 연산만을 위한 전처리

```
import tensorflow_datasets as tfds

datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
```

```
train_size = info.splits["train"].num_examples
test_size = info.splits["test"].num_examples
```

```
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
```

- 어휘사전 구축

```
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()))
```

- 전체 훈련 세트를 한 번 순회하면서 preprocess() 함수를 적용하고 Counter로 단어의 등장 횟수를 센다.
- 가장 많이 등장하는 단어 3개 확인

```
vocabulary.most_common()[:3]
```

```
[(b'<pad>', 214309), (b'the', 61137), (b'a', 38564)]
```

- 10000개만 남기고 삭제

```
vocab_size = 10000
truncated_vocabulary = [
    word for word, count in vocabulary.most_common()[:vocab_size]]
```

- 각 단어를 ID로 바꾸는 전처리(1000개의 oov 버킷을 이용하는 룩업테이블(2차원 테이블) 생성)

```
words = tf.constant(truncated_vocabulary)
word_ids = tf.range(len(truncated_vocabulary), dtype=tf.int64)
vocab_init = tf.lookup.KeyValueTensorInitializer(words, word_ids)
num_oov_buckets = 1000
table = tf.lookup.StaticVocabularyTable(vocab_init, num_oov_buckets)
```

- 단어 ID 확인

```
table.lookup(tf.constant([b"This movie was faaaaaantastic".split()]))
```

```
<tf.Tensor: shape=(1, 4), dtype=int64, numpy=array([[   22,    12,    11, 10053]])>
```

- 배치로 묶기 + 짧은 시퀀스로 바꾸기 + 인코딩 + 프리패치

```
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)
```

- 모델

```
embed_size = 128
model = keras.models.Sequential([
    keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size,
                           mask_zero=True, # not shown in the book
                           input_shape=[None]),
    keras.layers.GRU(128, return_sequences=True),
    keras.layers.GRU(128),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
history = model.fit(train_set, epochs=5)
```

- 첫번째 층 → 단어 ID를 임베딩으로 변환하는 Embedding 층
- 임베딩 행렬 : 단어 ID당 하나의 행 + 임베딩 차원당 하나의 열
- 모델의 입력 : [배치크기, 타임 스탭 수] → 2D 텐서
  Embedding 층의 출력 : [배치크기, 타임 스탭 수, 임베딩 크기] → 3D 텐서
- GPU층 2개로 구성
- 두번째 층 → 마지막 타임 스텝의 출력만 반환
- 출력층 : 시그모이드 함수 사용
- 긍정적인 감정을 표현하는지에 대한 추정 확률

▼ 1. 마스킹

- **<u><'pad'> 토큰 때문에 masking이라는 개념이 등장했다. padding된 토큰들은 실제 학습시, loss 계산을 하지 않기 위해 masking을 만들어서 불필요한 계산을 없앴다</u>**
- Embedding 층을 만들때 mask_zero=True 매개변수를 추가 ⇒ 패딩 토큰 무시

//위에도 있다

- 함수형 API를 이용하여 직접 마스킹을 처리

```
K = keras.backend
embed_size = 128
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.models.Model(inputs=[inputs], outputs=[outputs])
model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
history = model.fit(train_set, epochs=5)
```

▼ 2. 사전훈련된 임베딩 재사용하기

- 모듈 : 사전훈련된 모델 컴포넌트를 모델에 추가하기 쉽게 만들어주는 모델 컴포넌트

  (텐서플로 허브 프로젝트)

- nnlm-en-dim50 문장 임베딩 모듈 버전 1을 감정 분석 모델에 사용

```
import tensorflow_hub as hub

model = keras.Sequential([
    hub.KerasLayer("https://tfhub.dev/google/tf2-preview/nnlm-en-dim50/1",
                   dtype=tf.string, input_shape=[], output_shape=[50]),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam",
              metrics=["accuracy"])
```

- hub.KerasLayer 층이 주어진 URL에서 모듈(문장 인코더)을 다운로드
- 문자열을 입력으로 받아 하나의 벡터로 인코딩한다. 내부적으로는 문자열을 파싱해서 대규모 코퍼스에서 사전훈련된 임베딩 행렬을 사용해 각 단어를 임베딩한다.
- 다음 2개의 Dense 층을 추가해 감정분석모델을 만든다.
- 데이터셋을 다운로드 하면 바로 훈련할 수 있다.

```
import tensorflow_datasets as tfds

datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
batch_size = 32
train_set = datasets["train"].batch(batch_size).prefetch(1)
history = model.fit(train_set, epochs=5)
```

```
Epoch 1/5
782/782 [==============================] - 5s 5ms/step - loss: 0.5461 - accuracy: 0.7267
Epoch 2/5
782/782 [==============================] - 4s 5ms/step - loss: 0.5130 - accuracy: 0.7495
Epoch 3/5
782/782 [==============================] - 4s 5ms/step - loss: 0.5081 - accuracy: 0.7532
Epoch 4/5
782/782 [==============================] - 4s 5ms/step - loss: 0.5047 - accuracy: 0.7540
Epoch 5/5
782/782 [==============================] - 4s 5ms/step - loss: 0.5018 - accuracy: 0.7566
```

▼ 16.3 신경망 기계 번역을 위한 인코더-디코더 네트워크

▼ 신경망 기계 번역

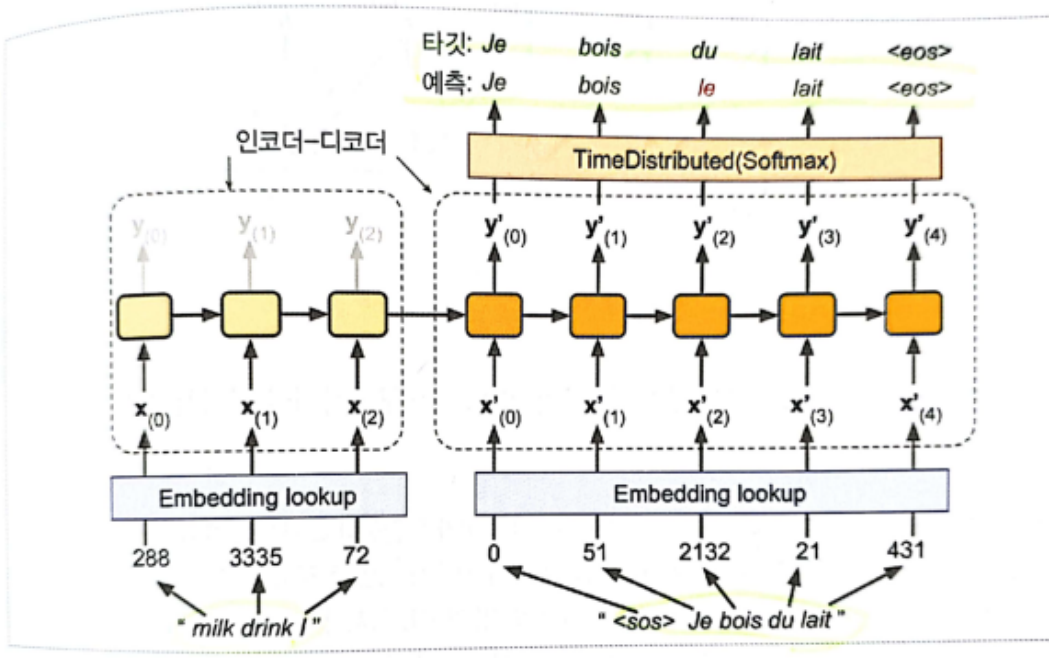


그림 16-3 간단한 신경망 기계 번역 모델

- 문제점

1. 문장의 길이가 다르다.
2. EOS 토큰 이후의 출력을 모두 무시한다.
3. 출력 어휘 사전이 방대할경우 매우 느려진다.

```
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)
sequence_lengths = keras.layers.Input(shape=[], 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)
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 = tfa.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.models.Model(
    inputs=[encoder_inputs, decoder_inputs, sequence_lengths],
    outputs=[Y_proba])
```

- p651~652 필자가 언급한 것

▼ 1. 양방향 RNN

- 양방향 순환 층 : 동일한 입력에 대해 두 개의 순환층을 실행, 하나는 왼쪽 → 오른쪽으로 단어를 읽고 다른 하나는 오른쪽 → 왼쪽으로 읽는다. 그 다음 일반적으로 타임 스텝마다 이 두 출력을 연결한다.

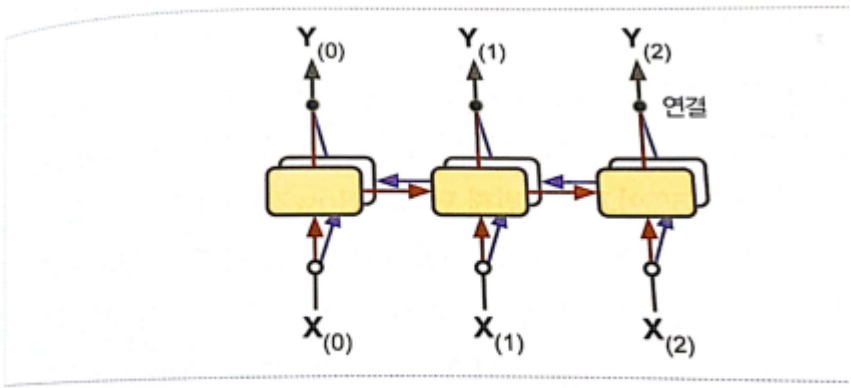


그림 16-5 양방향 순환 층

```
model = keras.models.Sequential([
    keras.layers.GRU(10, return_sequences=True, input_shape=[None, 10]),
    keras.layers.Bidirectional(keras.layers.GRU(10, return_sequences=True))
])
```

▼ 2. 빔 검색

- 빔 검색 : k개의 가능성있는 문장의 리스트를 유지하고 디코더 단계마다 이 문장의 단어를 하나씩 생성하여 가능성있는 k개의 문장을 만듭니다.
- 빔너비: 이때의 파라미터 k
- 계산 : 간단하게 단어의 확률을 구하고 조건부확률을 통해 문장의 확률을 계산하여 가장 높은 확률을 가진 문장을 추론(p653~654)

```
beam_width = 10
decoder = tfa.seq2seq.beam_search_decoder.BeamSearchDecoder(
    cell=decoder_cell, beam_width=beam_width, output_layer=output_layer)
decoder_initial_state = tfa.seq2seq.beam_search_decoder.tile_batch(
    encoder_state, multiplier=beam_width)
outputs, _, _ = decoder(
    embedding_decoder, start_tokens=start_tokens, end_token=end_token,
    initial_state=decoder_initial_state)
```

▼ 16.4 어텐션 메커니즘

▼ 어텐션

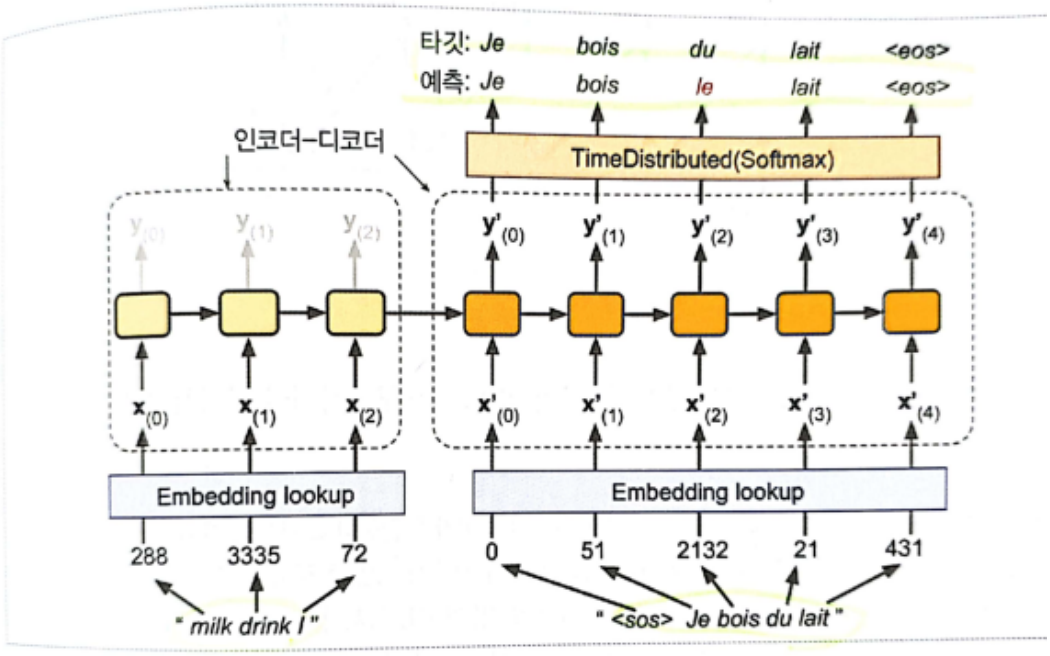


그림 16-3 간단한 신경망 기계 번역 모델

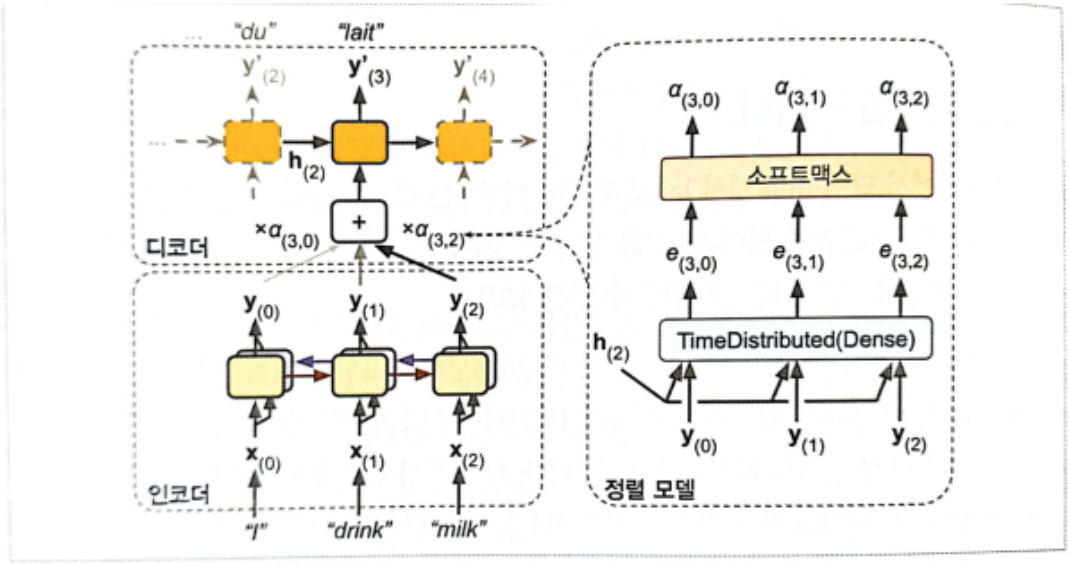


그림 16-6 어텐션 모델의 인코더-디코더 네트워크를 사용한 신경망 기계 번역

- **바흐다나우 어텐션 or 연결 어텐션 or 덧셈 어텐션**
- **루옹 어텐션**

식 16-1 어텐션 메커니즘

$$\tilde{\mathbf{h}}_{(t)} = \sum_i \alpha_{(t,i)} \mathbf{y}_{(i)}$$

여기에서

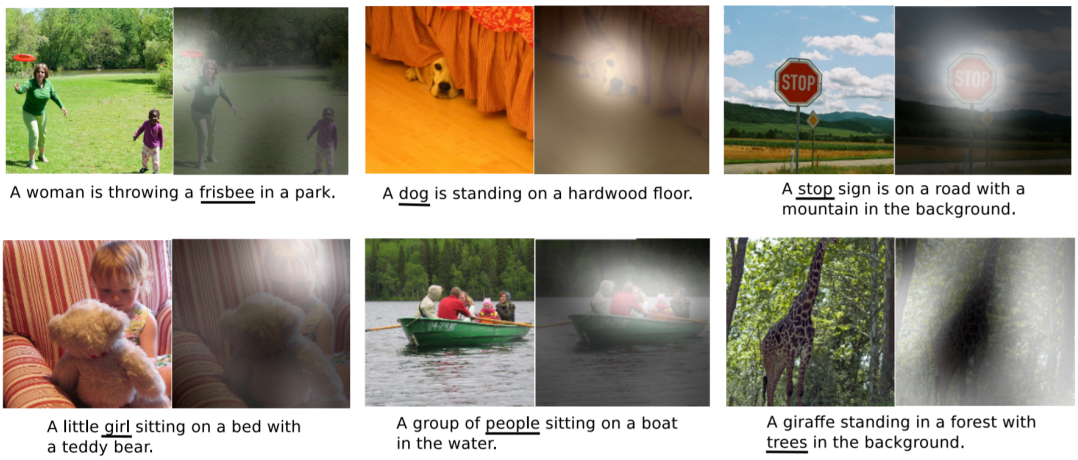
$$\alpha_{(t,i)} = \frac{\exp\left(e_{(t,i)}\right)}{\sum_{i'} \exp\left(e_{(t,i')}\right)} \text{ 이고}$$

$$e_{(t,i)} = \begin{cases} \mathbf{h}_{(t)}^{\intercal} \mathbf{y}_{(i)} & \text{점곱} \\ \mathbf{h}_{(t)}^{\intercal} \mathbf{W} \mathbf{y}_{(i)} & \text{일반 점곱} \\ \mathbf{v}^{\intercal} \tanh\left(\mathbf{W}\left[\mathbf{h}_{(t)}; \mathbf{y}_{(i)}\right]\right) & \text{연결} \end{cases}$$

- 자세한 내용은 p657

▼ 1. 비주얼 어텐션

- 비주얼 어텐션을 사용한 이미지 캡션 생성



- 입력 이미지가 있을때 어떤 부분에 초점을 맞추고 찍혔는지 알 수 있음

▼ 2. 트랜스포머 구조: 어텐션이 필요한 전부다

▼ 트랜스포머 구조

- 순환 층이나 합성곱 층을 전혀 사용하지 않고 어텐션 메커니즘만 사용

- NMT 문제에서 최고 수준의 성능을 크게 향상
- 장점: 빠르게 훈련, 병렬화하기 쉬움

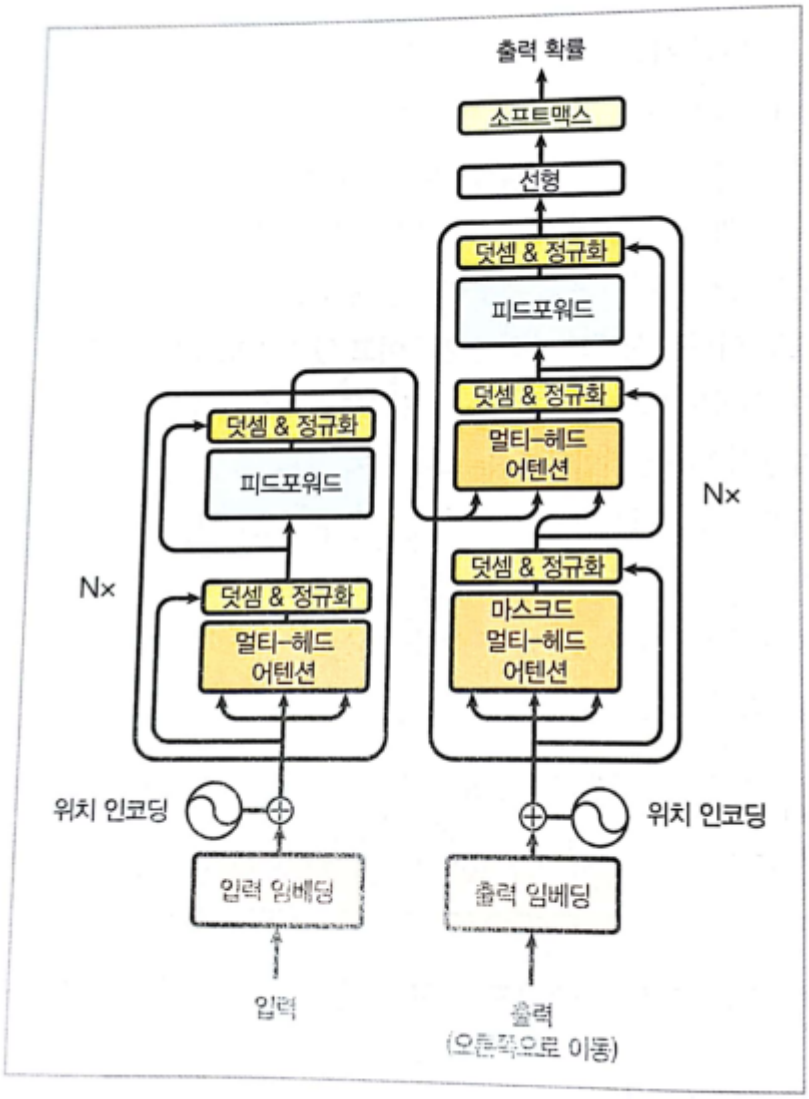


그림 16-8 트랜스포머 구조[31]

- p661

▼ 위치 인코딩

- 문장 안에 있는 단어의 위치를 인코딩한 밀집 벡터
- i번째 위치 인코딩이 문장에 있는 i번째 단어의 단어 임베딩에 더해진다.

식 16-2 사인/코사인 위치 인코딩

$$P_{p,2i} = \sin\left(p / 10000^{2i/d}\right)$$
$$P_{p,2i+1} = \cos\left(p / 10000^{2i/d}\right)$$

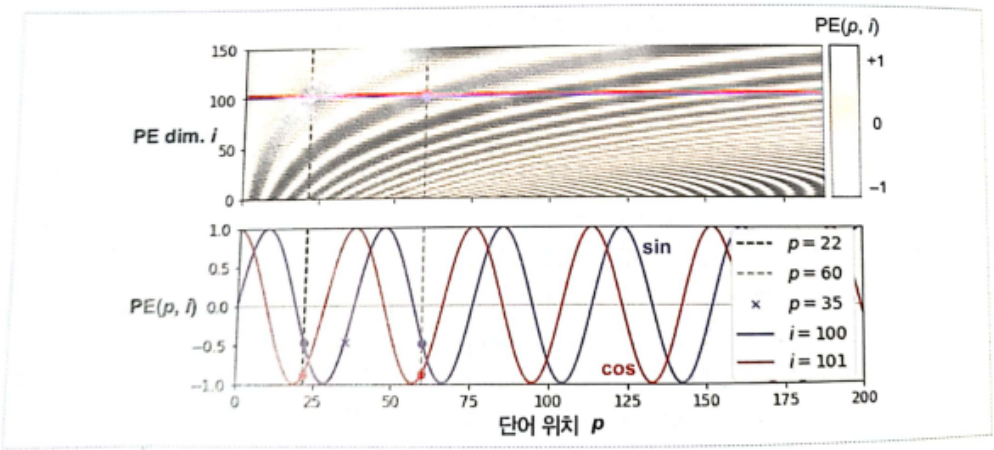


그림 16-9 (전치된) 사인/코사인 위치 인코딩 행렬(위쪽), 두 i값에 대한 그래프(아래쪽)

- 사인 코사인 함수로 정의한 고정된 위치 인코딩 선호
- 텐서플로에서 만들기

```
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 # max_dims must be even
        p, i = np.meshgrid(np.arange(max_steps), np.arange(max_dims // 2))
        pos_emb = np.empty((1, max_steps, max_dims))
        pos_emb[0, :, ::2] = np.sin(p / 10000**(2 * i / max_dims)).T
        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]]
```

```
embed_size = 512; max_steps = 500; vocab_size = 10000
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)
positional_encoding = PositionalEncoding(max_steps, max_dims=embed_size)
encoder_in = positional_encoding(encoder_embeddings)
decoder_in = positional_encoding(decoder_embeddings)
```

- 아래와같이 트랜스포머 모델을 간소화하여 만들 수 있음

```
Z = encoder_in
for N in range(6):
    Z = keras.layers.Attention(use_scale=True)([Z, Z])

encoder_outputs = Z
Z = decoder_in
for N in range(6):
    Z = keras.layers.Attention(use_scale=True, causal=True)([Z, Z])
    Z = keras.layers.Attention(use_scale=True)([Z, encoder_outputs])

outputs = keras.layers.TimeDistributed(
    keras.layers.Dense(vocab_size, activation="softmax"))(Z)
```

▼ 멀티 헤드 어텐션

▼ 스케일드 점-곱 어텐션

“They played chess”

{주어 : They, 동사: played, ...}

모델에서 키를 표현하기 위한 벡터 표현 ⇒ 쿼리

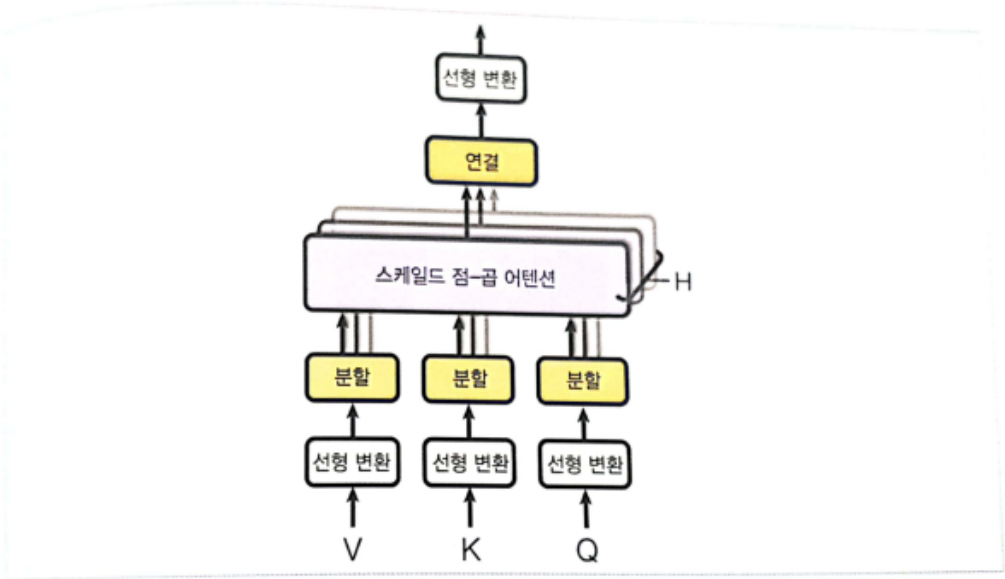


그림 16-10 멀티-헤드 어텐션 층 구조[35]

- 멀티 헤드 어텐션 = 스케일드 점-곱 어텐션 층의 묶음
- V: 행마다 값 하나를 담은 행렬 K: 행마다 키 하나를 담은 행렬 Q: 행마다 쿼리 하나를 담은 행렬
- 각 층은 값, 키, 쿼리의 선형 변환이 선행된다.
- 출력은 단순히 모두 연결되어 마지막 선형 반환을 통과한다.

▼ 16.5 언어 모델 분야의 최근 혁신

- **언어 모델 기반의 임베딩**
- **NLP 작업을 위한 비지도 사전훈련의 효과 검증**
- **마스크드 언어 모델(MLM)**
- **다음 문장 예측(NSP)**