자연어생성
Seq2Seq

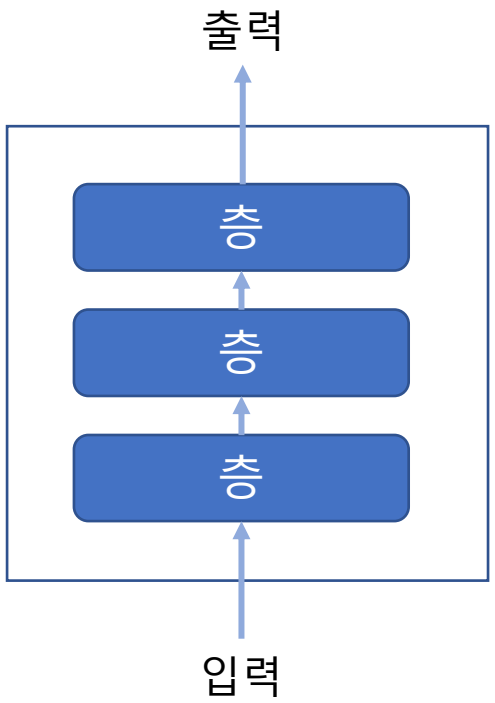
최 석 재

lingua@naver.com

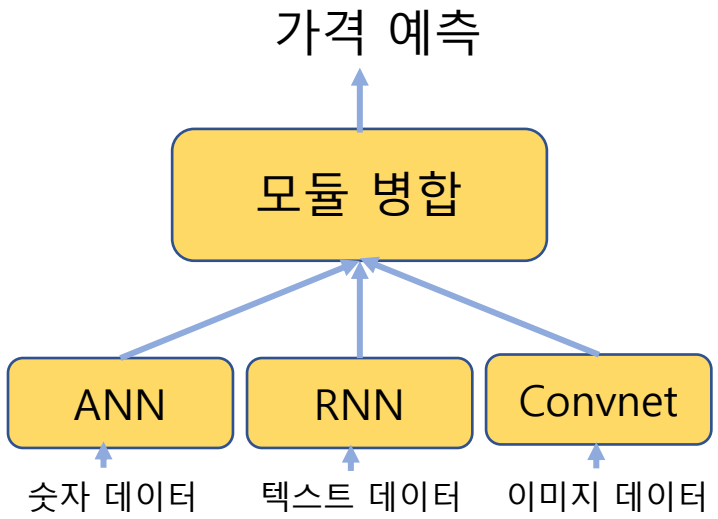
함수형 API

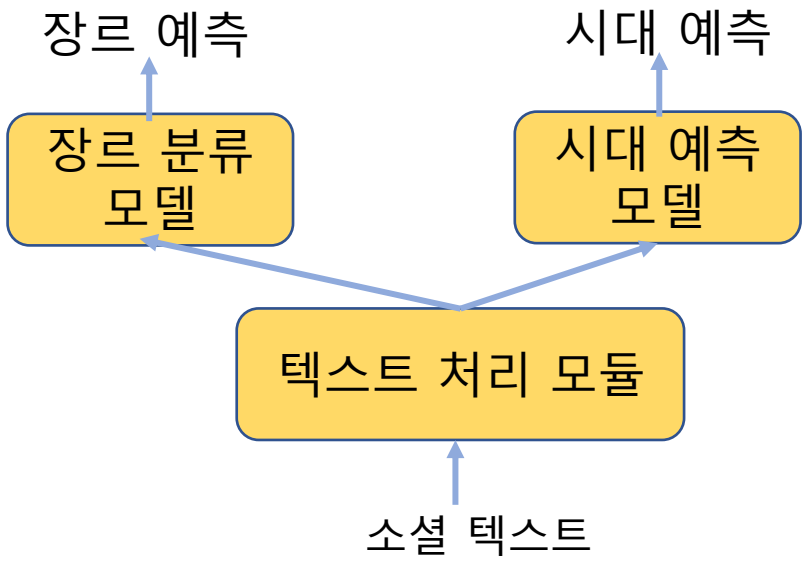
함수형 API *Functional API*

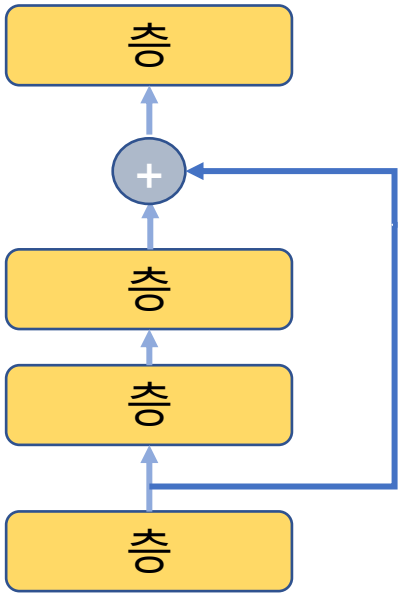
- Sequential 모델은 네트워크의 입력과 출력이 하나라고 가정한다



- 함수형 API는 다중입력, 다중출력, 그래프 등 다양한 구조를 지원한다







Sequential과 함수형 API 비교

- 단순한 모델로 Sequential과 Functional 을 비교한다
- Sequential 모델은 층을 하나하나 쌓아가면서 모델을 만드는 구조이다
- Functional API는 input에서 output으로 가는 과정을 설정하고,
  Model() 함수에 input과 output을 넣어 모델을 만드는 구조이다

Sequential

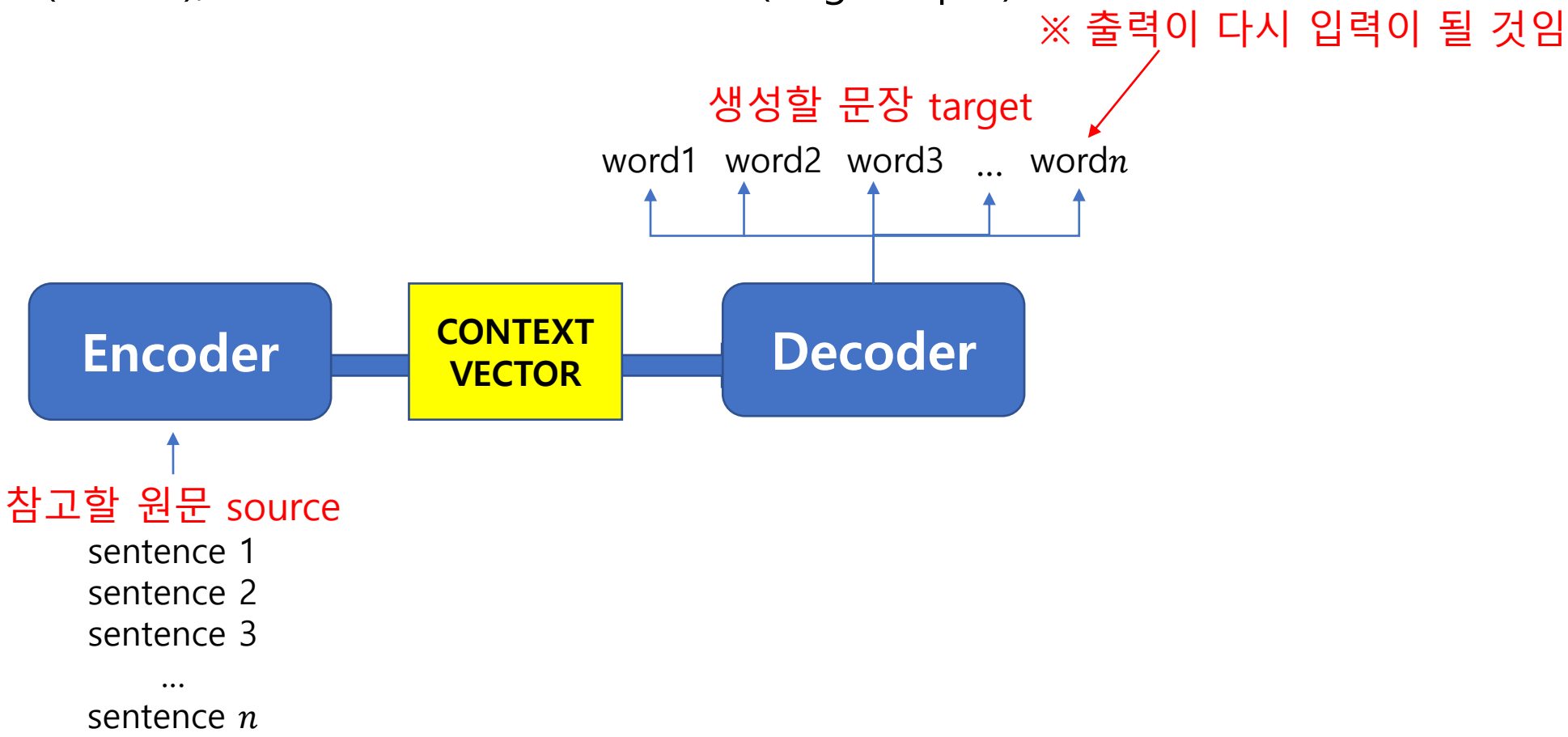
```
model = Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(64,)))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

Functional API

```
input_tensor = Input(shape=(64,))
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x)
model = Model(input_tensor, output_tensor)
```

번역 모델

- 번역 모델은 두 개의 언어이므로 두 개의 입력이 필요하다
- 하나는 원문(source), 다른 하나는 그 때의 번역문(target input)이다

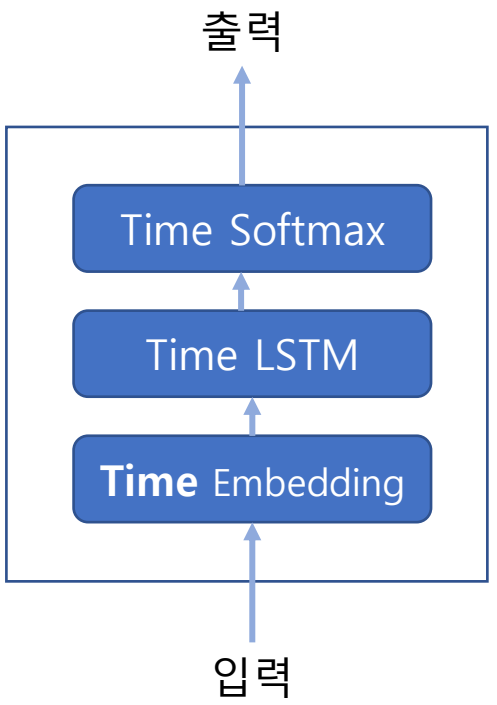


문장 생성

- 문장을 생성하는 기본적인 방법은 다음과 같다

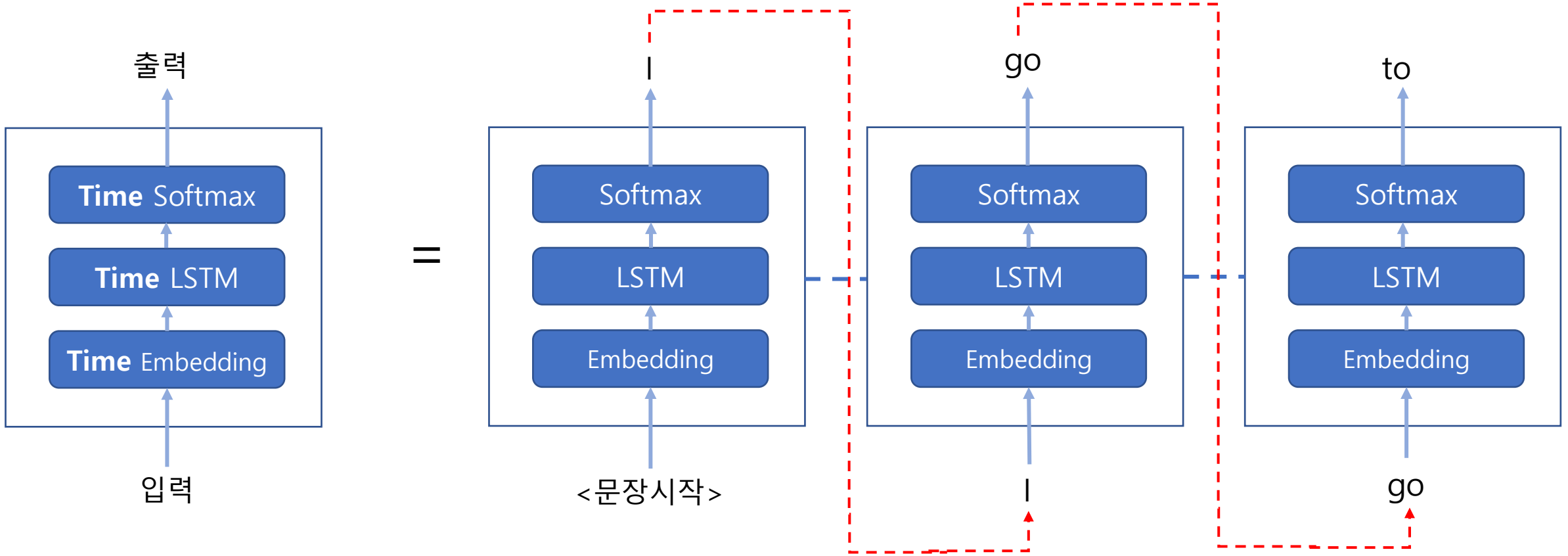
- "I go to a school", "I go to a library"와 같은 많은 문장을 훈련시킨다
- LSTM은 각 단어의 발생 확률을 단어의 순서에 따라 학습한다
- 위의 문장의 경우, "I" 뒤에는 "go", "go" 뒤에는 "to"의 발생확률이 높을 것이다

- 따라서 어떤 사람이 입력문의 시작으로 "I"를 넣으면
- LSTM은 망설이지 않고 "I go to a"까지를 자동으로 생성한다
- "go"를 넣으면 "go to a"를 자동 생성한다
- 그 뒤에 "school" 또는 "library"를 넣을지는 다른 분포를 더 봐야 한다



시계열 발생확률을 이용한 문장 생성

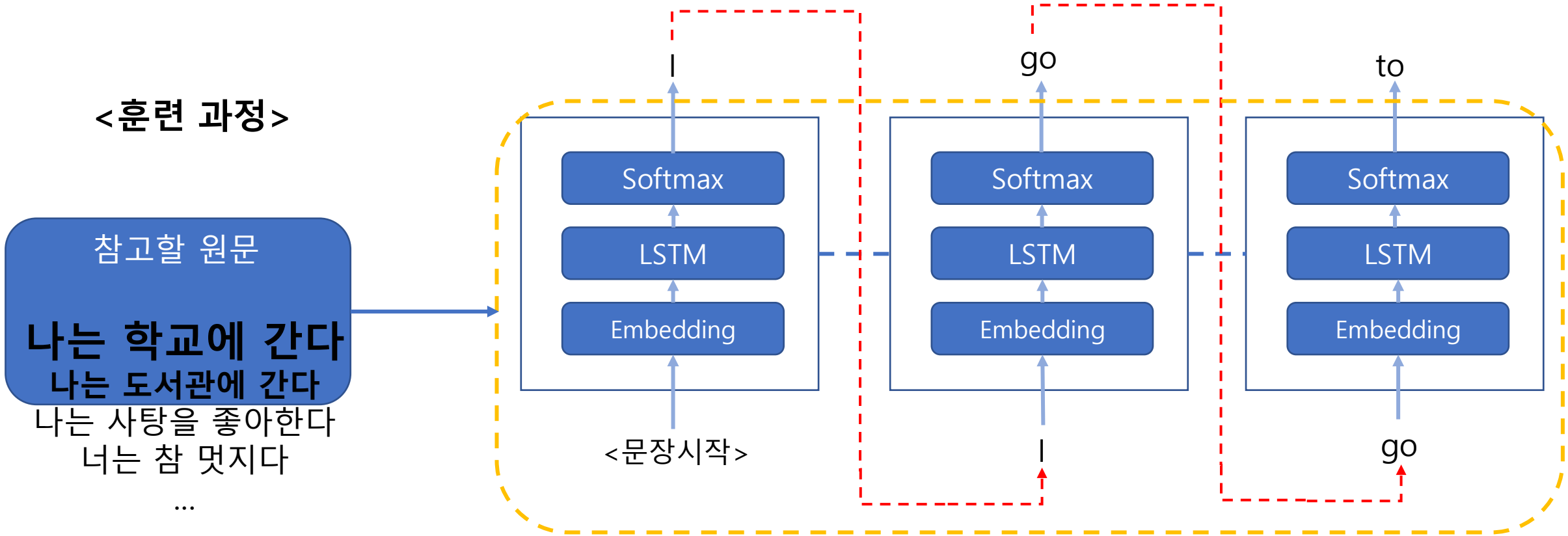
- 앞 타임스텝의 출력 결과로 다음 타임스텝에 발생할 단어를 예측한다



번역 모델을 훈련시킬 때
'나는 학교에 간다'와 'I go to a school' 두 개의 입력을 같이 훈련시킨다

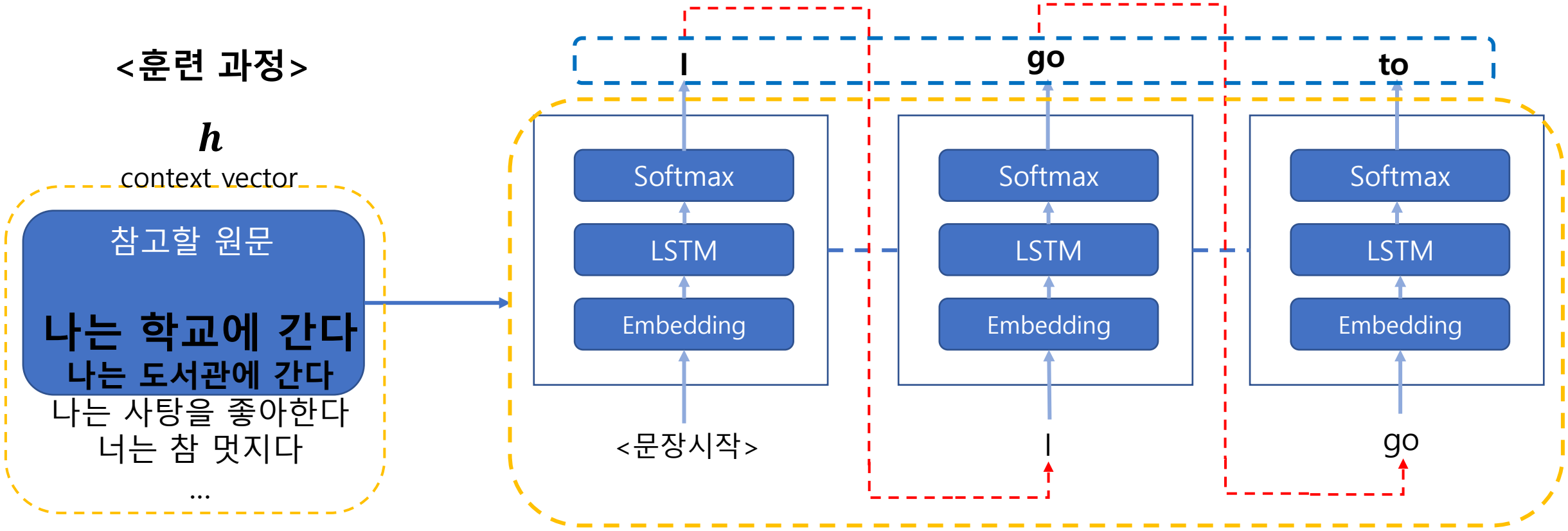
'나는 학교에 간다'와 유사한 문장이 많을 때 (특정 Embedding 값에 대한 높은 빈도)
입력된 문장이 '나는 학교에 간다'와 유사하다면 첫 문장의 시작은 출력으로 'I'가 나오도록 훈련된다
계속, '나는 학교에 간다' 類 문장에서 'I' 단어 다음에는 'go'가 나오도록 훈련된다
계속, '나는 학교에 간다' 類 문장에서 'go' 단어 다음에는 'to'가 나오도록 훈련된다

즉, 각 단어를 보고 단어 대 단어로 번역하는 것이 아니라, (그래서 두 언어 단어의 수와 순서는 관계없음)
문장을 자동 생성하되, 기존에 입력된 문장과 유사한 문장을 찾아 자동 생성하도록 훈련된다



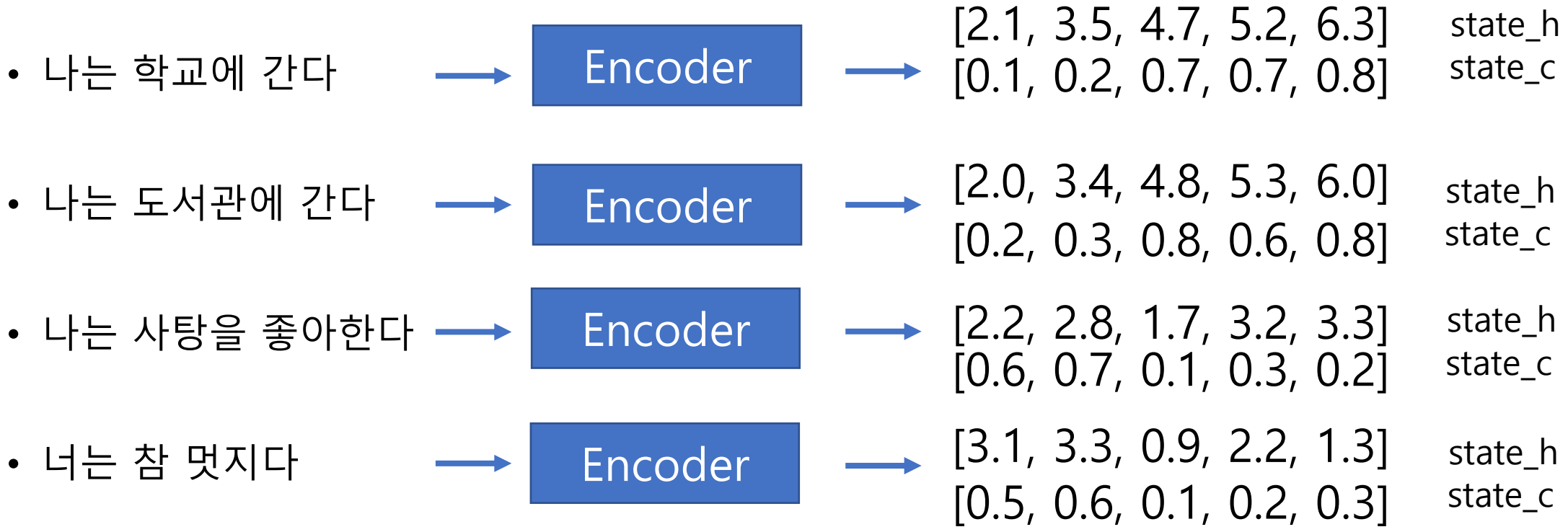
훈련 과정에서는 번역문 출력 부분인 'I', 'go', 'to', ... 를 미리 알려준다
즉, '나는 학교에 간다' 類 문장이 오면 첫 시작은 'I',
'나는 학교에 간다' 類 문장에서 앞 단어 출력이 'I'이면 두 번째 단어 출력은 'go',
'나는 학교에 간다' 類 문장에서 앞 단어 출력이 'go'이면 다음 단어 출력은 'to'
라고 알려주며 계속 훈련해 간다

번역 모델은 참고할 원문을 텍스트 형태로 가지고 있는 것이 아니라
숫자로 기호화된 벡터로 변환해 가지고 있다
이를 Context Vector 라고 한다



Context Vector의 모습

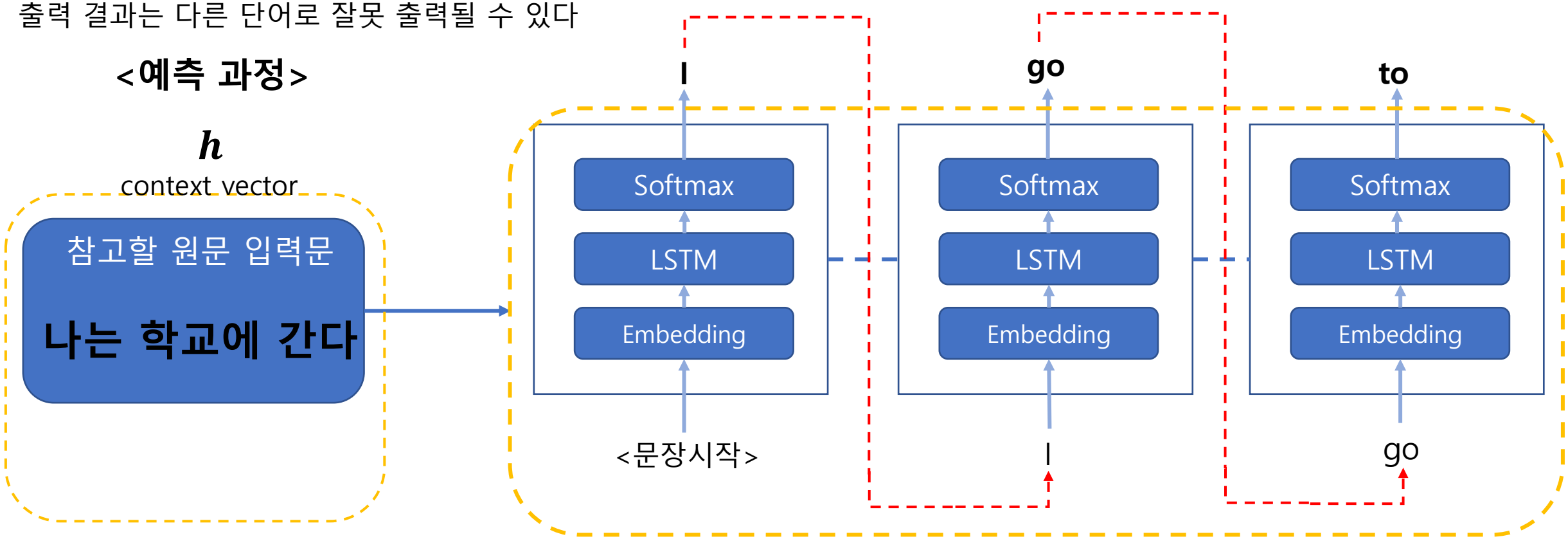
- 원문은 Encoder를 거쳐 고정된 길이의 벡터로 출력된다



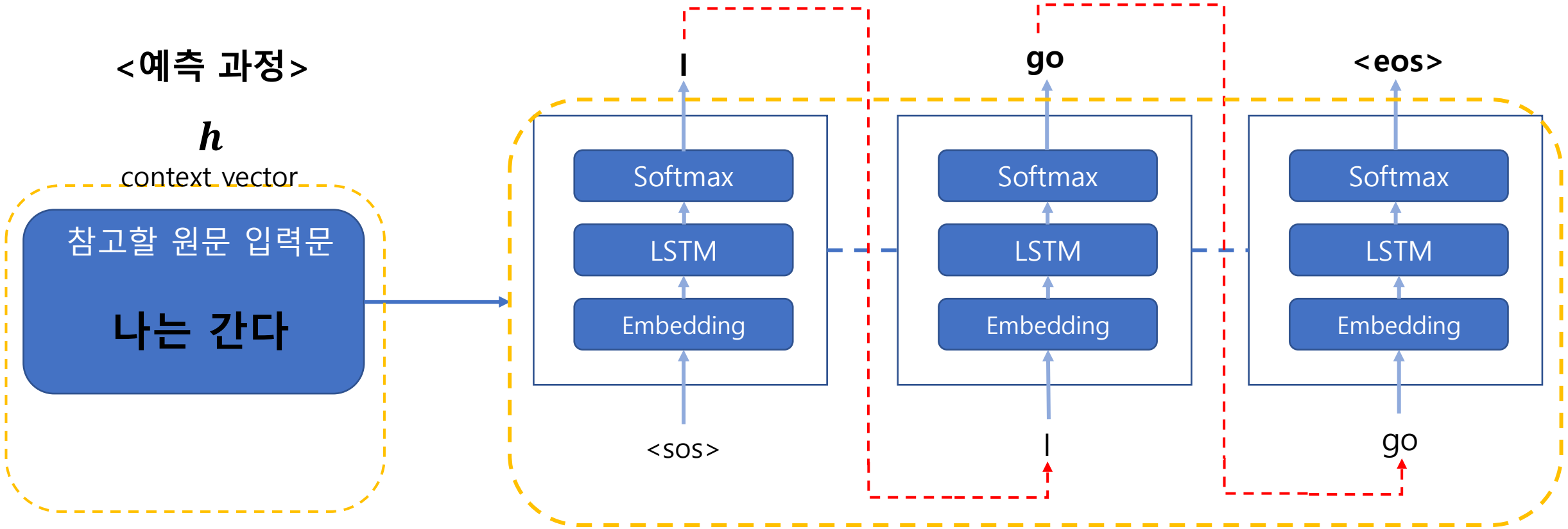
예측 과정에서는 당연히 번역문의 출력이 무엇이 되어야 하는지 알려줄 수 없다
단지 번역 대상인 원문만 가지고 예측을 진행한다

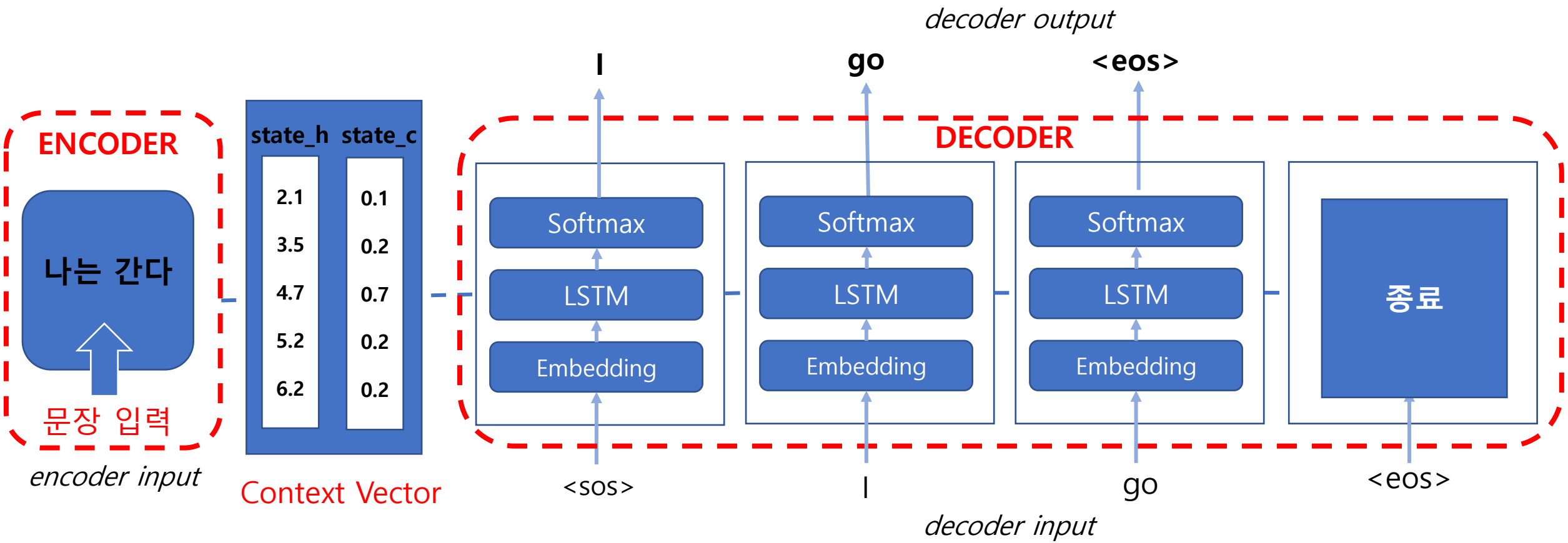
번역 대상 원문이 입력되면
모델은 숫자 벡터로 변환된 원문 입력문과 시작 기호의 인덱스 값으로 문장 생성을 시작한다
'나는 학교에 간다' 類 문장에서 문장의 첫 시작은 'I'를 출력하도록 훈련되었다
'나는 학교에 간다' 類 문장에서 'I' 다음에는 'go'가 출력되도록 훈련되었다
'나는 학교에 간다' 類 문장에서 'go' 다음에는 'to'가 출력되도록 훈련되었다

예측시 입력된 문장은 훈련시 입력된 문장들과는 조금씩은 다를 것이므로
출력 결과는 다른 단어로 잘못 출력될 수 있다



처음 시작은 시작 기호(예: <sos>, '₩t')를 입력하는 것으로 시작하여
종료 기호(예: <eos>, '₩n')가 예측 출력되면 문장 생성과정이 끝난다





입력 문장을 받아 부호화 하는 부분은 encoder,
숫자로 부호화된 것을 자연어로 출력하는 부분은 decoder

```
[array([[-0.05642597, 0.01665201, 0.02717819, 0.05304213, 0.01755996, -0.00242471, -0.00054484, -0.04550469, -0.00982542, 0.01023033, -0.03469724, -
0.02518192, -0.0210219 , 0.03386844, -0.0342838 , 0.03300365, -0.04286133, 0.025323 , -0.0223102 , 0.03730808, -0.0352733 , -0.04017608, 0.00225591, -
0.00061181, 0.02576207, 0.01868865, -0.00809146, 0.00488537, -0.04350194, 0.03120992, 0.06443618, 0.01070095, 0.04250181, 0.02224795, 0.04098651,
0.04862582, 0.00576775, -0.02113362, 0.02428788, -0.0406853 , -0.04925001, -0.03004065, -0.00710994, 0.04086966, -0.01403381, -0.01587017, -0.0351746 ,
-0.02972642, -0.01148238, 0.01676985, 0.004784 , 0.02736723, -0.02040982, -0.01115751, -0.01636275, -0.03341633, 0.03628072, 0.02514911, 0.00926209,
0.03609961, -0.0074698 , 0.00381904, 0.02144871, 0.04743378, 0.03065111, -0.03056311, -0.01166071, -0.03995139, -0.00375476, -0.03967547, -0.02159382,
0.00263612, -0.00146384, -0.02855268, 0.01928439, 0.02586632, -0.00841798, -0.01521898, -0.05746694, 0.03762262, 0.01192692, 0.02833915, 0.00764597, -
0.0292516 , 0.08694846, -0.00127408, -0.03893797, 0.0294005 , -0.05631784, -0.01911976, -0.0461707 , 0.02018022, 0.01559 , 0.06184876, 0.01037537, -
0.01218646, -0.00017321, 0.01025015, 0.03952975, 0.05874261, 0.03098417, 0.02684566, -0.02001139, -0.03220342, -0.03191935, 0.00517929, 0.03111773,
0.00048311, -0.04695559, -0.03252606, 0.02271254, 0.02038911, -0.00360776, 0.01708099, 0.00576047, 0.03393069, 0.09005471, -0.02006661, 0.02677193, -
0.042997 , -0.00421534, 0.00701342, -0.00628152, -0.03085191, 0.02327253, 0.00218514, -0.01528222, -0.015867 , -0.01100116, -0.05046576, 0.01505716, -
0.01340475, -0.01535769, 0.03125777, 0.00585154, -0.00645894, 0.01857192, 0.02000132, 0.01738971, -0.03789321, -0.06007411, 0.03956808, 0.02606883,
0.03533803, -0.00778296, -0.02512777, 0.02427388, -0.02656788, -0.03096194, -0.05489504, 0.02368894, -0.04389361, 0.00883204, 0.00790481, 0.07556886,
0.00324968, -0.02673473, 0.03483502, -0.0001514 , 0.00494187, 0.05903387, -0.00963618, -0.02862738, -0.0272558 , -0.02942557, -0.02442834, 0.03065128,
0.03870518, 0.00750464, -0.00242943, 0.0227034 , -0.02475173, -0.0187747 , -0.00283365, -0.00859974, 0.00210751, -0.04015836, 0.03441415, -0.05292362,
0.00357471, 0.05304396, 0.0137649 , 0.00621216, 0.05573941, 0.01762769, -0.02165108, 0.04156377, 0.01438216, -0.02326358, -0.05806401, -0.03681963,
0.03526486, 0.03067053, 0.01001206, -0.03291085, -0.03709491, 0.03613593, 0.01613604, -0.05027019, 0.03524018, -0.05204295, 0.01524146, -0.00291917,
0.03790669, 0.01030064, -0.02827318, 0.03010313, 0.00172787, -0.00238671, 0.05546506, 0.0133656 , -0.00229286, 0.02791926, 0.01148778, 0.03615043,
0.02123334, -0.01230029, -0.03318655, 0.04697248, -0.00537674, 0.02909049, 0.05611795, 0.03183028, -0.03346087, 0.02002454, 0.01049229, 0.00021555, -
0.04719242, 0.01129786, -0.00925096, -0.02851911, -0.00211228, 0.04804622, -0.01197799, -0.00821385, -0.01601935, -0.00343577, -0.0111023 , -0.05112459,
-0.03783128, 0.04164566, 0.00785125, -0.03805341, 0.00655445, -0.03879622, 0.03241379, 0.02933786, 0.01520632, 0.00290501, -0.02307232, 0.0071147 ,
0.01522603, 0.00554983, 0.02962036, 0.01449001, -0.00081743]], dtype=float32), array([[-0.11223224, 0.03500225, 0.05367855, 0.10607582, 0.0341643 , -
0.00496991, -0.00108073, -0.09329987, -0.01997304, 0.01972717, -0.06858329, -0.05187633, -0.042285 , 0.06544399, -0.07074782, 0.064735 , -0.08371538,
0.05278783, -0.0438807 , 0.0747695 , -0.06869103, -0.07896521, 0.00455747, -0.00125519, 0.05180836, 0.03729143, -0.01625618, 0.00996056, -0.08440381,
0.0626603 , 0.12478738, 0.02207588, 0.08656372, 0.04431285, 0.08127625, 0.09948211, 0.01180881, -0.04137007, 0.04933755, -0.08397454, -0.09833883, -
0.05962639, -0.01509414, 0.07881491, -0.02838221, -0.03311141, -0.07056791, -0.0585319 , -0.0237176 , 0.03452192, 0.00980378, 0.05585844, -0.03995125, -
0.02311863, -0.03210365, -0.06733871, 0.07000028, 0.04981825, 0.01915049, 0.07257982, -0.01519741, 0.00761291, 0.04476855, 0.09266956, 0.05917979, -
0.05862471, -0.02305559, -0.07821333, -0.00771742, -0.08033659, -0.04444728, 0.0051106 , -0.00282932, -0.0572776 , 0.03794549, 0.05170341, -0.01740045,
-0.0315838 , -0.10956806, 0.07294266, 0.02481589, 0.05762612, 0.01596532, -0.0594636 , 0.17020166, -0.00252928, -0.07707605, 0.05846139, -0.10988025, -
0.03805165, -0.0917358 , 0.04054455, 0.03177618, 0.12386961, 0.02139368, -0.02425599, -0.00035647, 0.02130676, 0.07706515, 0.12057655, 0.06102967,
0.0558782 , -0.04030226, -0.06276129, -0.06637 , 0.01025914, 0.0612295 , 0.00101466, -0.09677815, -0.06481004, 0.04692782, 0.04237395, -0.00755189,
0.03514173, 0.01124897, 0.06856917, 0.17582616, -0.03994364, 0.05256414, -0.08659571, -0.00851227, 0.013729 , -0.01256002, -0.06214024, 0.04515119,
0.00433286, -0.0306542 , -0.03039481, -0.0225775 , -0.09924375, 0.03057135, -0.02730249, -0.03064379, 0.06112061, 0.01159807, -0.01247253, 0.03760063,
0.04083059, 0.03353469, -0.07766038, -0.11777594, 0.08231935, 0.05150364, 0.07122753, -0.01619861, -0.04954065, 0.05027484, -0.05239422, -0.06210205, -
0.10997805, 0.05001988, -0.08835296, 0.01822096, 0.01661858, 0.14929053, 0.00651451, -0.05420889, 0.06876607, -0.00031046, 0.00980426, 0.12027006, -
0.01983368, -0.0604533 , -0.05402138, -0.05936119, -0.04978631, 0.06007272, 0.07866267, 0.0156952 , -0.0049569 , 0.04444938, -0.0519499 , -0.03646122, -
0.00587044, -0.01762376, 0.00405205, -0.07750569, 0.06932581, -0.10166039, 0.00692034, 0.10665835, 0.02825584, 0.01200644, 0.10747348, 0.03513418, -
0.04413404, 0.08209641, 0.02874271, -0.04542945, -0.11682425, -0.0719851 , 0.07002749, 0.0623176 , 0.02000209, -0.0639247 , -0.07486308, 0.07406417,
0.03270172, -0.10222199, 0.07152192, -0.10586092, 0.03212831, -0.00614403, 0.07532401, 0.02080558, -0.05636225, 0.05975332, 0.00349369, -0.00471065,
0.1109678 , 0.02665323, -0.00445687, 0.05559158, 0.02317693, 0.07204372, 0.042610
0.10778211, 0.06560112, -0.06764015, 0.04063585, 0.02221893, 0.00045109, -0.0902
0.02350643, -0.01656935, -0.03250483, -0.00669305, -0.02278086, -0.10192351, -0
0.06428701, 0.05817505, 0.03144236, 0.00566371, -0.04746522, 0.01442672, 0.0304
```

"Run!" 문장에 대한 state_h와 state_c

(붉은색 부분은 state_h, 검은색 부분은 state_c)

seq2seq 실습

음절 기반

seq2seq 기본 자료

- seq2seq 기본 코드로 잘 알려진 프랑소아 숄레(케라스 개발자)의 코드를 기본으로 진행한다
- 음절 기반의 코드로서 word embedding은 사용하지 않는다
- 음절기반으로서 예측되는 형태의 종류는 제한적이다
- 음절 기반은 한 어절 내의 음절 연쇄를 예측하는 것이므로 어색한 어절이 만들어질 수 있다

- 케라스 개발자 프랑수아 숄레의 블로그
- https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html

- 연습용 병렬 코퍼스 모음
- http://www.manythings.org/anki/

Colab에서의 경로 설정

```
# from google.colab import auth
# auth.authenticate_user()
```

- from google.colab import drive
- drive.mount('/content/gdrive')

```
# 경로 설정
```

- path = '/content/gdrive/My Drive/pytest/eng-kor/'
- !ls '/content/gdrive/My Drive/pytest/eng-kor/'

```
eng-kor_small.txt  eng-kor.txt
```

데이터 확인

빠른 진행을 위해 small 데이터를 사용한다

읽은 데이터는 DataFrame 이다

- import pandas as pd
- data = pd.read_csv(path+'eng-kor_small.txt', names=['source', 'target'], sep='₩t', encoding='utf-8')
- print('data length:', len(data))
- print('data type:', type(data))
- print('data shape:', data.shape)
- print('data sample:₩n', data.sample(5))

```
data length: 1000
data type: <class 'pandas.core.frame.DataFrame'>
data shape: (1000, 2)
data sample:
              source     target
115     Birds fly.  새가 날고 있네.
127     Get ready.       준비해.
35         We won.   우리가 이겼어.
322  Look around.       둘러봐.
233   I am human.   나는 인간이야.
```

한 컬럼 확인

한 개 컬럼의 타입은 1차원 배열 형태인 Pandas Series이다

- print('data.target length:', len(data.target))
- print('data.target type:', type(data.target))
- print('data.target shape:', data.target.shape)
- print('data.target sample:₩n', data.target.sample(5))

```
data.target length: 1000
data.target type: <class 'pandas.core.series.Series'>
data.target shape: (1000,)
data.target sample:
 908     모든 것이 변했어.
721      모두가 웃었어.
60          그만해.
269      톰이 동의했어.
390    현실적으로 생각해!
Name: target, dtype: object
```

시작부호와 종료부호 부착

데이터가 모두 3종이 필요하다. source 언어에 encoder_input 1개, target 언어에 decoder_input, decoder_target 2개
encoder는 source 언어를 그대로 사용한다
decoder input 데이터의 시작에는 ₩t, 문장의 끝에는 ₩n를 부착한다 (음절기반이므로 여기서는 ₩t를 <SOS>, ₩n를 <EOS>로 사용)
decoder target 데이터는 ₩n만 필요하다 (₩n이 생성되면 생성과정 종료)

- data.target_input = data.target.apply(lambda x : '₩t'+x+'₩n')
- data.target_target = data.target.apply(lambda x : x+'₩n')
- print('₩ndata.target_input:₩n', data.target_input)
- print('₩ndata.target_target:₩n', data.target_target)

```
data.target_input:
 0                ₩t가.₩n
1               ₩t안녕.₩n
2               ₩t뛰어!₩n
3               ₩t뛰어.₩n
4               ₩t누구?₩n
              ...
995     ₩t노래하는 거 좋아해요?₩n
996      ₩t노래하는 거 좋아해?₩n
997    ₩t고양이를 좋아하지 않아?₩n
998      ₩t꿈은 이루어질 거야.₩n
999     ₩t모두 그녀를 사랑한다.₩n
Name: target, Length: 1000, dtype: object
```

```
data.target_target:
 0                 가.₩n
1                안녕.₩n
2                뛰어!₩n
3                뛰어.₩n
4                누구?₩n
              ...
995     노래하는 거 좋아해요?₩n
996      노래하는 거 좋아해?₩n
997    고양이를 좋아하지 않아?₩n
998      꿈은 이루어질 거야.₩n
999     모두 그녀를 사랑한다.₩n
Name: target, Length: 1000, dtype: object
```

타입 확인

apply 함수 적용 결과 타입에는 변함이 없다

- print('data.target_input length:', len(data.target_input))
- print('data.target_input type:', type(data.target_input))
- print('data.target_input shape:', data.target_input.shape)
- print('data.target_input sample:₩n', data.target_input.sample(5))

```
data.target_input length: 1000
data.target_input type: <class 'pandas.core.series.Series'>
data.target_input shape: (1000,)
data.target_input sample:
 774       ₩t이 자리 비었어₩n
418        ₩t이거 내꺼니?₩n
789    ₩t우린 경고 받았었어.₩n
991       ₩t영어 좋아해요?₩n
792         ₩t뭐가 좋아?₩n
Name: target, dtype: object
```

문장의 길이 maxlen 설정하기

Padding에 사용할 문장의 길이를 구한다. 여기서는 최대값으로 maxlen을 설정하였다
source 문장의 최대 길이를 구한다
source 문장의 최대 음절 길이로 maxlen을 설정

- max_src_len = data.source.apply(lambda x: len(x)).max()
- print('source sentence max length: ', max_src_len)

target 문장의 최대 길이를 구한다
target 문장의 최대 음절 길이로 maxlen을 설정

- max_tar_len = data.target_input.apply(lambda x: len(x)).max()-2
- print('target sentence max length: ', max_tar_len)

※ "₩t, ₩n"의 길이는 제외
len('₩t') == 1

```
source sentence max length:  20
target sentence max length:  19
```

Data Tokenizing

토크나이징으로 각 문자 종류에 숫자값을 배당한다

- from keras.preprocessing.text import Tokenizer

※ 음절 기반이므로 char_level=True를 사용한다
※ 케라스 토크나이저는 기본값으로 각종 문장부호를 제거한다
filters 아규먼트에서 조절할 수 있다

source 언어 Tokenizing

- tokenizer_source = Tokenizer(num_words=None, char_level=True, lower=False)
- tokenizer_source.fit_on_texts(data.source) # 인덱스를 구축한다
- word_index_source = tokenizer_source.word_index # 글자와 인덱스의 쌍을 가져온다

- print('₩n전체에서 %s개의 고유한 토큰을 찾았습니다.' % len(word_index_source))
- print('word_index_source: ', word_index_source)

```
전체에서 64개의 고유한 토큰을 찾았습니다.
word_index_source:  {' ': 1, 'e': 2, 'o': 3, '.': 4, 'a': 5, 't': 6, 'i': 7, 's': 8, 'n': 9, 'r': 10,
```

target 언어 Tokenizing

target 언어의 Tokenizer도 target_input으로만 만들면 된다 (target의 output은 같은 언어이므로)

- tokenizer_target = Tokenizer(num_words=None, char_level=True, lower=False)
- tokenizer_target.fit_on_texts(data.target_input) # 인덱스를 구축한다
- word_index_target = tokenizer_target.word_index # 글자와 인덱스의 쌍을 가져온다

- print('₩n전체에서 %s개의 고유한 토큰을 찾았습니다.' % len(word_index_target))
- print('word_index_target: ', word_index_target)

```
전체에서 558개의 고유한 토큰을 찾았습니다.
word_index_target: {' ': 1, '₩t': 2, '₩n': 3, '.': 4, '어': 5, '이': 6, '톰': 7, '해': 8, '아': 9, '은': 10, '다': 11,
```

1로 시작 시작부호 종료부호

Data Sequencing

배당된 숫자를 이용하여 각 문장의 문자를 숫자로 치환한다
source 언어 Sequencing

- encoder_input = tokenizer_source.texts_to_sequences(data.source)

- print('₩nResult of encoder_input sequencing: ')
- print(data.source[0], encoder_input[0])
- print(data.source[1], encoder_input[1])
- print(data.source[2], encoder_input[2])
- print(data.source[3], encoder_input[3])

```
Result of encoder_input sequencing:
Go. [38, 3, 4]
Hi. [32, 7, 4]
Run! [49, 16, 9, 30]
Run. [49, 16, 9, 4]
```

```
전체에서 64개의 고유한 토큰을 찾았습니다.
word_index_source:  {' ': 1, 'e': 2, 'o': 3, '.': 4, 'a': 5, 't': 6, 'i': 7, 's': 8, 'n': 9, 'r': 10,
```

유사한 문장

target 언어 Sequencing. input과 target에 같은 토크나이저 객체를 사용한다

- decoder_input = tokenizer_target.texts_to_sequences(data.target_input)
- decoder_target = tokenizer_target.texts_to_sequences(data.target_target)

- print('₩nResult of decoder_input sequencing: ')
- print(data.target_input[0], decoder_input[0])
- print(data.target_input[1], decoder_input[1])
- print(data.target_input[2], decoder_input[2])

- print('₩nResult of decoder_target sequencing: ')
- print(data.target_target[0], decoder_target[0])
- print(data.target_target[1], decoder_target[1])
- print(data.target_target[2], decoder_target[2])

※ decoder_input과 decoder_target에도
유사하거나 동일한 문장이 중복되어 있을 수 있다

```
Result of decoder_input sequencing:
₩t     가.₩n
[2, 13, 4, 3]
        안녕.
[2, 51, 223, 4, 3]
        뛰어!
[2, 272, 5, 18, 3]

Result of decoder_target sequencing:
가.     ₩n
[13, 4, 3]
안녕.
[51, 223, 4, 3]
뛰어!
[272, 5, 18, 3]
```

※ ₩t는 decoder_input에만 있다

타입 확인

토크나이징의 결과는 리스트 타입이다

- print('data.source type:', type(data.source)) # Series
- print('encoder_input type:', type(encoder_input)) # list

- print('data.source:₩n', data.source)
- print('encoder_input₩n:', encoder_input)

```
data.source type: <class 'pandas.core.series.Series'>
encoder_input type: <class 'list'>
data.source:
 0                      Go.
1                      Hi.
2                     Run!
3                     Run.
4                     Who?
              ...
995    Do you like singing?
996    Do you like singing?
997    Don't you like cats?
998    Dreams do come true.
999    Everybody loves her.
Name: source, Length: 1000, dtype: object
encoder_input
: [[38, 3, 4], [32, 7, 4], [49, 16, 9, 30], [49, 16, 9, 4], [31, 14, 3, 28],
```

Data Padding

- from keras.preprocessing.sequence import pad_sequences
- encoder_input = pad_sequences(encoder_input, maxlen=max_src_len, padding='post')
- decoder_input = pad_sequences(decoder_input, maxlen=max_tar_len, padding='post')
- decoder_target = pad_sequences(decoder_target, maxlen=max_tar_len, padding='post')

- print('₩nPadding result sample: ')
- print(data.target_input[0], decoder_input[0])

Padding 결과 샘플

```
Padding result sample:
        가.
[ 2 13  4  3  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
```

2	13	4	3
₩t	가	.	₩n

타입 확인

Padding의 결과는 Numpy 배열이다

- print('decoder_input length:', len(decoder_input))
- Print('decoder_input type:', type(decoder_input))
- print('decoder_input shape:', decoder_input.shape)

```
decoder_input length: 1000
decoder_input type: <class 'numpy.ndarray'>
decoder_input shape: (1000, 19)
```

※ 1000개 문장이며, 각 문장의 길이는 19인 2D 배열

One-Hot Encoding

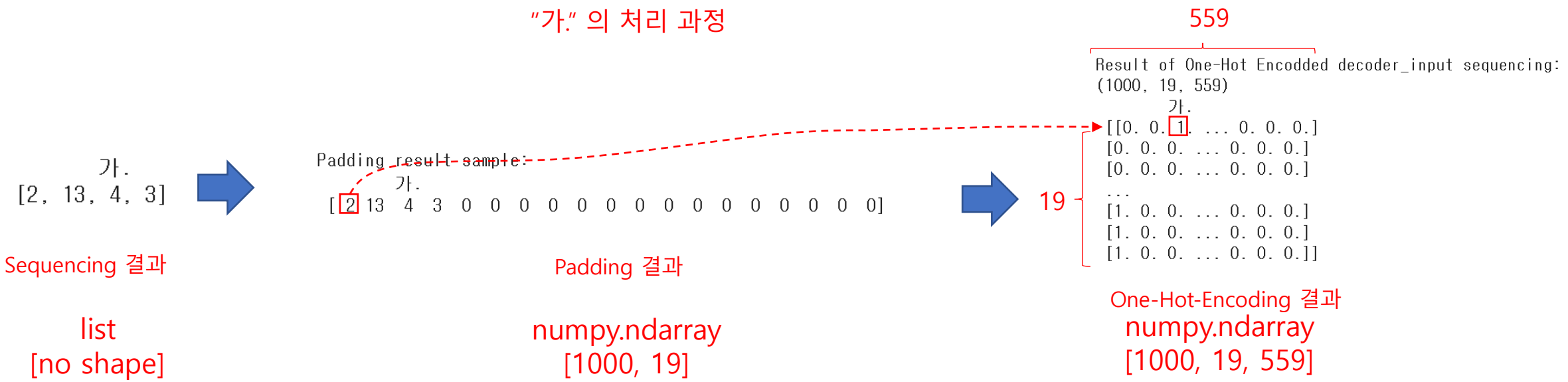
케라스 원-핫 인코딩을 수행한다
클래스의 수는 1을 올려주어야 한다 (Padding으로 생긴 0을 추가로 받아야 함)
케라스 원-핫 인코딩은 차원을 하나 더 늘린다

- from tensorflow.keras.utils import to_categorical
- encoder_input = to_categorical(encoder_input, num_classes=len(word_index_source)+1)
- decoder_input = to_categorical(decoder_input, num_classes=len(word_index_target)+1)
- decoder_target = to_categorical(decoder_target, num_classes=len(word_index_target)+1)

텍스트 처리 과정

결과를 출력하고, "Va !" 의 예로 이제까지의 처리 과정을 살펴본다

- print('₩nResult of One-Hot Encodded decoder_input sequencing: ')
- print(decoder_input.shape) # (1000, 19, 559). 1000개 문장이며, 각 문장 길이는 19(음절), 음절의 종류는 559
- print(data.target_input[0], decoder_input[0]) # 0번 문장 출력

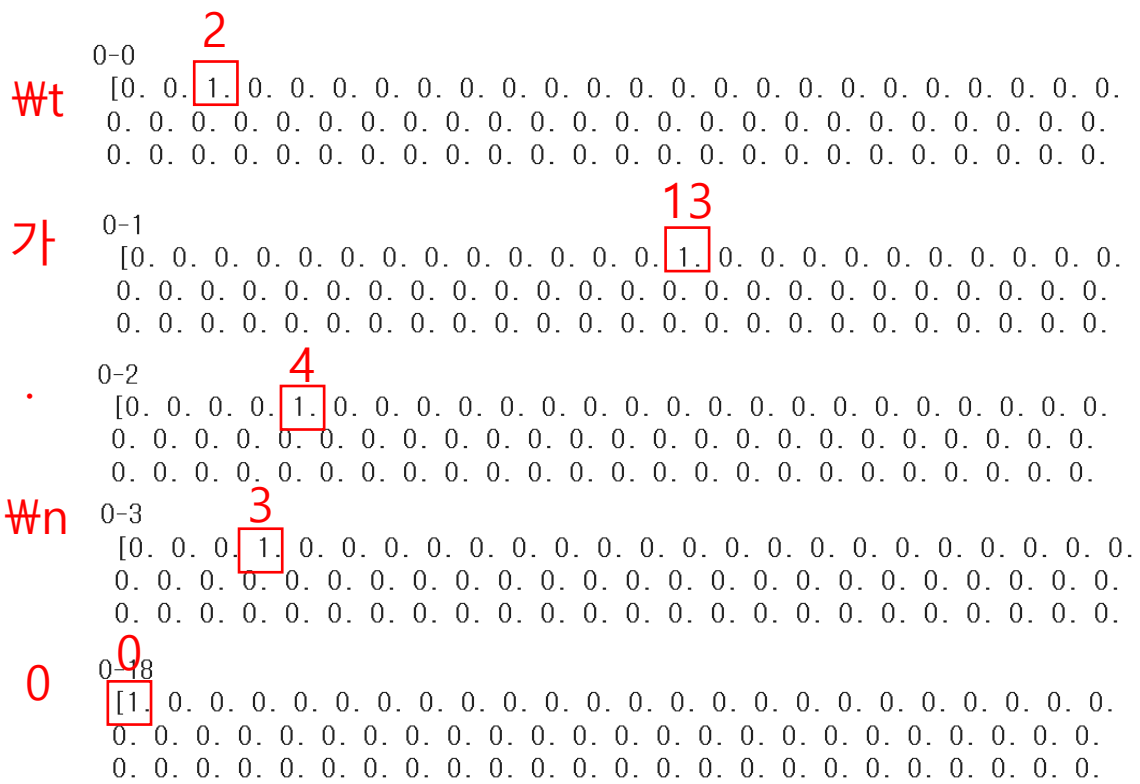


decoder_input[0] 출력

2	13	4	3
₩t	가	.	₩n

하나의 문장 '가.'를 갖고 있는 decoder_input[0]를 살펴본다

- print('0-0₩n', decoder_input[0][0])
- print('₩n0-1₩n', decoder_input[0][1])
- print('₩n0-2₩n', decoder_input[0][2])
- print('₩n0-3₩n', decoder_input[0][3])
- print('₩n0-18₩n', decoder_input[0][18])



각 문장은 19개의 음절로 Padding 되어 있고,
각 음절에서 0~558까지의 종류에 대하여
One-Hot Encoding이 되어 있다

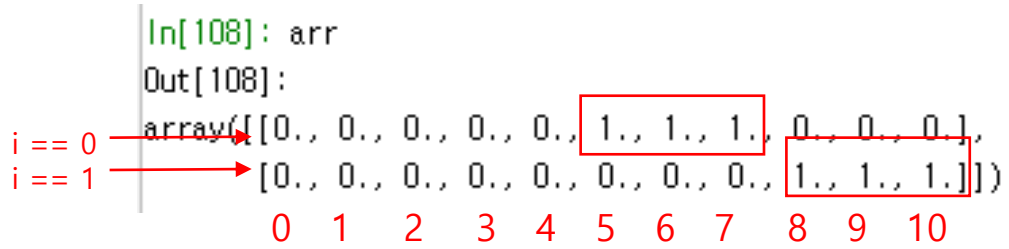
즉, 각 음절 자리마다 음절 종류가 새로운 차원이 되어
2D → 3D가 되었다

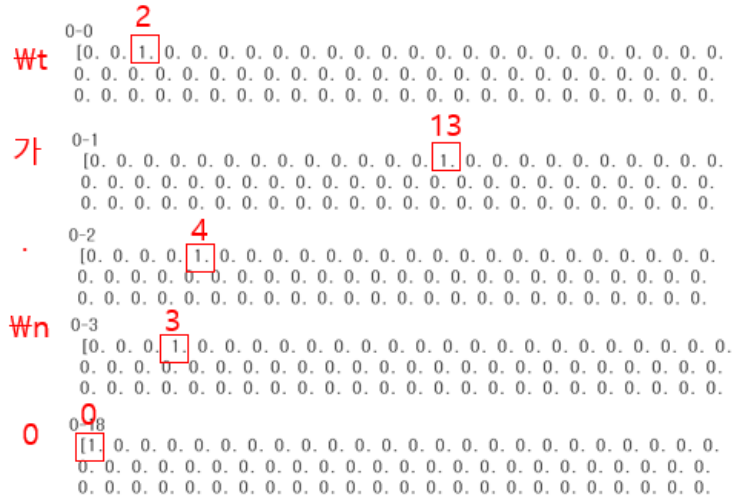
```
전체에서 558개의 고유한 토큰을 찾았습니다.
word_index_target:  {' ': 1, '₩t': 2, '₩n': 3, '.': 4, '어': 5, '이': 6, '톰': 7, '해': 8, '아': 9, '은': 10, '다': 11,
```

수동과 케라스 원-핫 인코딩의 차이

- 순환신경망까지는 수동으로 원-핫 인코딩 함수를 만들어 사용하였고,
- 여기서는 케라스의 원-핫 인코딩 함수를 사용하였다

- 수동 원-핫 인코딩은 여러 개의 1이 하나의 배열에 있는 방식이고,
- 케라스의 원-핫 인코딩은 하나의 배열에는 하나의 1만 있는 방식이다





타입 확인

케라스 원-핫 인코딩으로 차원이 늘어 결과는 3D이다

- print('decoder_input length:', len(decoder_input))
- print('decoder_input type:', type(decoder_input))
- print('decoder_input shape:', decoder_input.shape)

```
decoder_input length: 1000
decoder_input type: <class 'numpy.ndarray'>
decoder_input shape: (1000, 19, 559)
```

※ 1000 문장, 각각의 음절 길이는 19에, 각 음절은 559개의 차원 특성을 가짐

Teacher Forcing

- **교사 강요를 이용한 모델 훈련**
- 예측 과정에서는 이전 시점의 decoder_input 예측 결과를 현재 시점의 decoder_input에 넣는다
- 그러나 훈련 과정에서 그렇게 하면 잘못 예측된 이전 시점의 결과가 현재 시점으로 들어간다
- 훈련 과정에서는 이전 시점 decoder_input의 실제값(정답)을 현재 시점의 decoder_input에 넣는다
- 이를 교사 강요라고 한다

훈련용 Encoder

- from keras.models import Model
- from keras.layers import Input, LSTM, Dense

 # Encoder – Source

입력문의 길이는 문장마다 다르므로 None. 원-핫 인코딩 출력 결과는 len(word_index_source)+1

- encoder_inputs = Input(shape=(None, len(word_index_source)+1))

encoder 내부 상태(은닉상태, 셀상태)를 decoder로 넘겨주기 위해 return_state=True

- encoder_lstm = LSTM(units=256, return_state=True)

return_state=True로 만들어진 모델이므로 은닉상태와 셀상태를 받는다. encoder_outputs는 사용하지 않는다

- encoder_outputs, state_h, state_c = encoder_lstm(encoder_inputs)
- encoder_states = [state_h, state_c]

훈련용 Decoder

 # Decoder – Input

- decoder_inputs = Input(shape=(None, len(word_index_target)+1))

 # Decoder – Output. decoder의 첫 상태를 encoder의 은닉상태와 셀상태로 한다

256 unit으로 된 encoder_states를 받아야 하므로 decoder의 은닉 노드도 256으로 encoder에 맞춘다

- decoder_lstm = LSTM(units=256, return_sequences=True, return_state=True)

모든 타임스텝의 결과를 출력

내부상태(state_h, state_c)를 받기 위해

Decoder의 은닉상태와 셀상태는 훈련 과정에서는 사용하지 않는다(_)

- decoder_outputs, _, _ = decoder_lstm(decoder_inputs, initial_state=encoder_states)
- decoder_dense = Dense(len(word_index_target)+1, activation='softmax')
- decoder_outputs = decoder_dense(decoder_outputs)

출력층의 크기도 번역문의 글자(또는 단어)가 가질 수 있는 크기로 한다(len(word_index_target)+1)

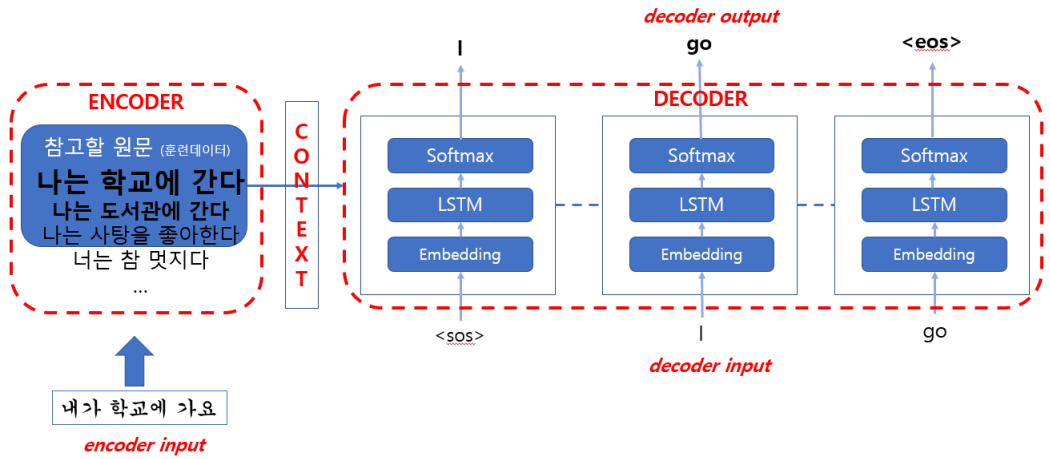
모델 훈련

epochs는 50회 정도 주어야 제대로 결과가 나온다

여기서는 일단 1회 훈련 때의 결과를 본다

- model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
- model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
- model.fit(x=[encoder_input, decoder_input], y=decoder_target, batch_size=64, epochs=1, validation_split=0.2)

```
13/13 [==============================] - 4s 76ms/step - loss: 3.4246 - val_loss: 3.1664
<keras.callbacks.History at 0x7f32fb0ebc10>
```

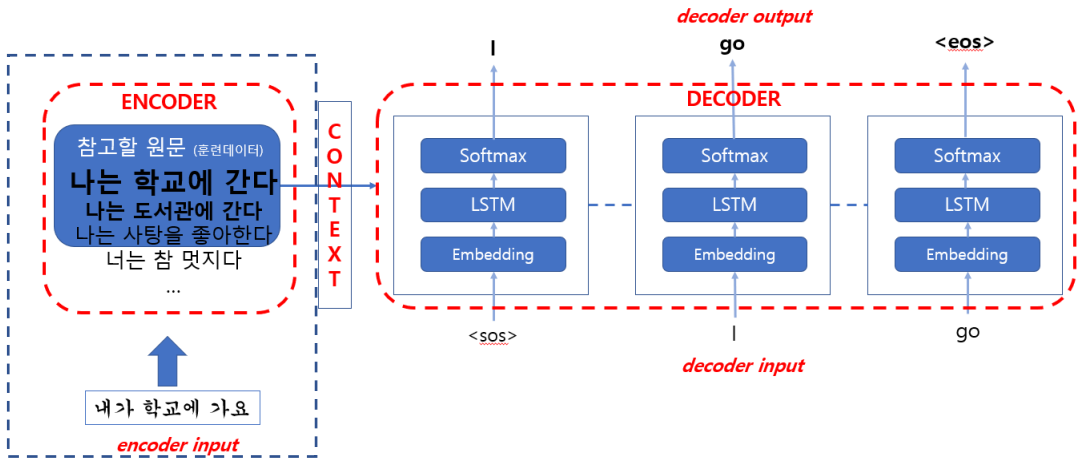


※ encoder_inputs와 decoder_inputs, decoder_outputs는 앞에서 만든 구조, encoder_input, decoder_input, decoder_target은 실제 데이터

예측용 Encoder

입력된 문장을 인코더에 넣어서 은닉상태와 셀상태를 얻는다
encoder_inputs, encoder_states는 훈련용에서 구성한 것을 사용한다
훈련용 Encoder에서 정의한 대로 encoder_inputs에서 encoder_states 출력의 과정이 연결된다
생성된 모델에 predict() 함수를 사용하여 encoder_states를 받는다
단순히 입력문에 대하여 상태 벡터를 얻는 것이므로 모델을 새로 구성할 필요가 없다

- encoder_model = Model(inputs=encoder_inputs, outputs=encoder_states)



예측용 Decoder

- decoder_state_input_h = Input(shape=(256,))
- decoder_state_input_c = Input(shape=(256,))

Encoder Context Vector의 두 내부 상태를 받기 위한 정의
(훈련용 Decoder는 Teacher Forcing으로 정의하지 않았음)

- decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]

훈련용 Decoder에서 구성한 것을 사용

- decoder_outputs, state_h, state_c = decoder_lstm(decoder_inputs, initial_state=decoder_states_inputs)

다음 예측을 위해 은닉상태와 셀상태를 받는다

- decoder_states = [state_h, state_c]
- decoder_outputs = decoder_dense(decoder_outputs)

출력층의 크기는 번역문의 글자(또는 단어)가 가질 수 있는 크기 (decoder_outputs의 shape == len(word_index_target)+1)

- decoder_model = Model(inputs=[decoder_inputs] + decoder_states_inputs, outputs=[decoder_outputs] + decoder_states)

Model의 inputs로는 target_input과 현재의 문장상태를 넣고,
outputs로는 예측 결과 및 은닉상태와 셀상태가 나오게 한다

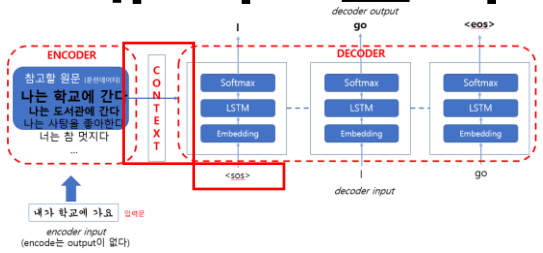
index – word Dictionary 구성

word로부터 index를 얻은 것을 index로부터 word를 얻는 것으로 바꿈

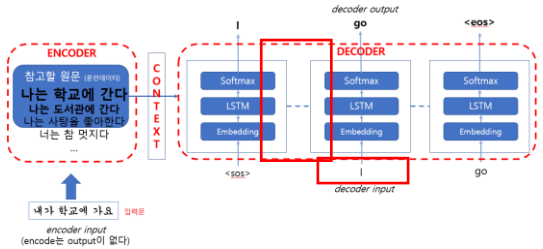
- index_to_src = dict((i, char) for char, i in word_index_source.items())
- index_to_tar = dict((i, char) for char, i in word_index_target.items())
- print(index_to_tar)

```
{1: ' ', 2: '₩t', 3: '₩n', 4: '.', 5: '어', 6: '이', 7: '톰', 8: '해', 9: '아', 10: '은', 11: '다',
```

예측 함수




```
def decode_sequence(input_seq):
    states_value = encoder_model.predict(input_seq)                  # 입력문을 인코더에 넣어 초기 문장상태벡터를 얻는다
    target_seq = np.zeros((1, 1, len(word_index_target)+1))          # 디코더 초기화
    target_seq[0, 0, word_index_target['₩t']] = 1.       # 디코더의 첫 시작은 <₩t>이므로 ₩t 위치에 원-핫 인코딩으로 기록
    stop_condition = False
    decoded_sentence = ""

    while not stop_condition:                                        # stop_condition이 True가 될 때까지 반복
        output_tokens, h, c = decoder_model.predict([target_seq] + states_value) # target_input과 문장상태벡터 입력
        sampled_token_index = np.argmax(output_tokens)               # 가장 큰 값을 갖는 인덱스를 선택

        if(sampled_token_index==0):                          # 0번 음절은 없으므로(Padding), 예측==0은 공백 음절 1로 치환한다
            sampled_token_index = 1

        sampled_char = index_to_tar[sampled_token_index]             # index-word dictionary를 이용해 단어 추출
        decoded_sentence += sampled_char                             # 추출된 단어를 앞의 결과와 연결

        if (sampled_char == '₩n' or len(decoded_sentence) > max_tar_len):    # 예측 종료 조건 (₩n 또는 최대길이)
            stop_condition = True

        target_seq = np.zeros((1, 1, len(word_index_target)+1))      # target_input 초기화
        target_seq[0, 0, sampled_token_index] = 1.                   # 직전 예측 결과를 원-핫 인코딩으로 입력
        states_value = [h, c]                                        # 문장상태벡터 업데이트
    return decoded_sentence
```



예측 과정

- import numpy as np
- for seq_index in [1, 2, 3]: # 입력 문장의 인덱스
- input_seq = encoder_input[seq_index:seq_index+1] # 3차원 배열에서는 이와 같이 [n:n+1] 형태로 해주어야 3차원이 유지되면서 n번째가 출력된다
- decoded_sentence = decode_sequence(input_seq)

- print(35 * "-")
- print('입력 문장:', data.source[seq_index])
- print('정답 문장:', data.target[seq_index][1:len(data.target[seq_index])])
- print('번역기가 번역한 문장:', decoded_sentence[:len(decoded_sentence)-1]) # '₩n'은 빼고 출력

```
-----------------------------------
입력 문장: Hi.
정답 문장: 안녕.
번역기가 번역한 문장:
-----------------------------------
입력 문장: Run!
정답 문장: 뛰어!
번역기가 번역한 문장:
-----------------------------------
입력 문장: Run.
정답 문장: 뛰어.
번역기가 번역한 문장:
```

Epochs에 따른 성능 향상

epochs = 10

```
----------------------------------
입력 문장: Hi.
정답 문장: 안녕.
번역기가 번역한 문장: 톰     .
----------------------------------
입력 문장: Run!
정답 문장: 뛰어!
번역기가 번역한 문장: 이
----------------------------------
입력 문장: Run.
정답 문장: 뛰어.
번역기가 번역한 문장: 이    .
```

epochs = 50

```
----------------------------------
입력 문장: Hi.
정답 문장: 안녕.
번역기가 번역한 문장: 가.
----------------------------------
입력 문장: Run!
정답 문장: 뛰어!
번역기가 번역한 문장: 안녕!
----------------------------------
입력 문장: Run.
정답 문장: 뛰어.
번역기가 번역한 문장: 가.
```

epochs = 100

```
----------------------------------
입력 문장: Hi.
정답 문장: 안녕.
번역기가 번역한 문장: 안녕.
----------------------------------
입력 문장: Run!
정답 문장: 뛰어!
번역기가 번역한 문장: 점프!
----------------------------------
입력 문장: Run.
정답 문장: 뛰어.
번역기가 번역한 문장: 뛰어.
```

1개 문장 예측하기

텍스트 데이터를 텐서에 넣기 위해서는 앞에서와 같이 다음의 과정을 거쳐야 한다

- Tokenizing (tokenizer_source.texts_to_sequences)
- Padding (pad_sequences)
- One-Hot Encoding (to_categorical)

1개 문장 예측하기

```
# 전처리
input_seq_1 = tokenizer_source.texts_to_sequences([list('wait!')])
input_seq_1 = pad_sequences(input_seq_1, maxlen=max_src_len, padding='post')
input_seq_1 = to_categorical(input_seq_1, num_classes=len(word_index_source)+1)

# 예측하기
decoded_sentence = decode_sequence(input_seq_1)
print(decoded_sentence)
```

싸!

한-영 번역

- 음절 기반으로 번역을 하는 이유는 데이터가 부족하기 때문이다
- 한국어의 경우 총 11,172개의 음절이 있다
- 즉, 한국어 번역시 매번 11,172 개 중 하나의 음절을 예측해야 한다

- 하지만 단어 기반 번역을 하려면 훨씬 많은 종류가 필요하다
- 보통 각 언어는 30만 개 정도의 단어를 가지고 있다고 본다
- 30만 개 중 하나의 단어를 예측해야 하므로 매우 많은 데이터가 필요하다

- 어절을 단어로 간주하면 한국어의 경우에는 매우 많은 종류의 어절이 있다
- 조사와 어미가 덧붙어 비슷한 의미의 많은 어절이 존재한다

연습문제

- kor-eng 폴더의 kor-eng.txt 파일을 이용하여 seq2seq를 수행하시오 (easy)

- epochs 5 훈련에서는 다음과 같은 결과가 출력된다

```
-----------------------------------
입력 문장: 나는 일어나자마자 화장실에 가요.
정답 문장: I go to bathroom as soon as I wake up.
번역기가 번역한 문장: I want to take a company to the work today.
-----------------------------------
입력 문장: 내년에 말레이시아로 연수를 가요.
정답 문장: I will be trained in Malaysia next year.
번역기가 번역한 문장: I will go to the same at the day of the day.
-----------------------------------
입력 문장: 나는 애슐리와 아웃백을 자주 가요.
정답 문장: I often go to outback with Ashley.
번역기가 번역한 문장: I want to take a company to the work today.
-----------------------------------
입력 문장: 도서관도 많이 다닐 수 있을 것 같아.
정답 문장: It is likely that I would be going to the library a lot.
번역기가 번역한 문장: I will take a contact restaurant for the same time.
```