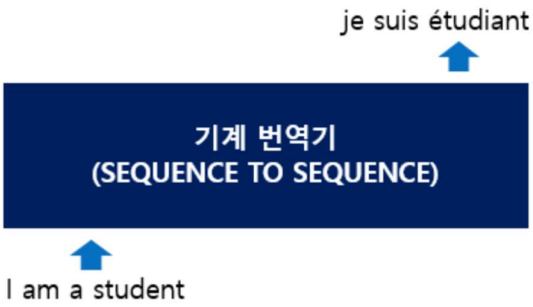
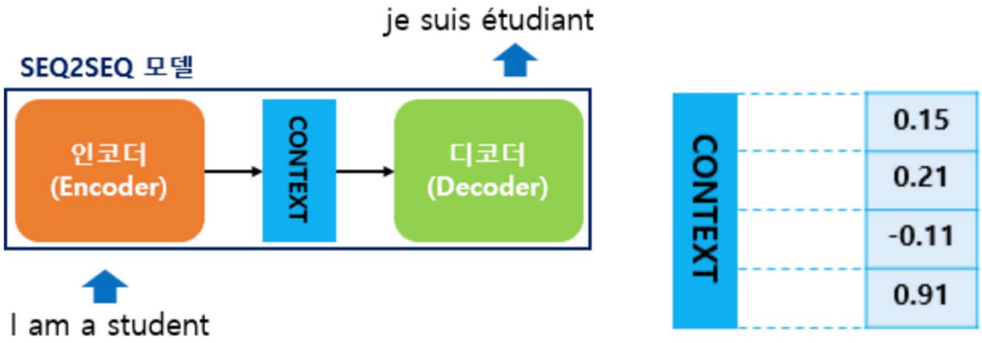
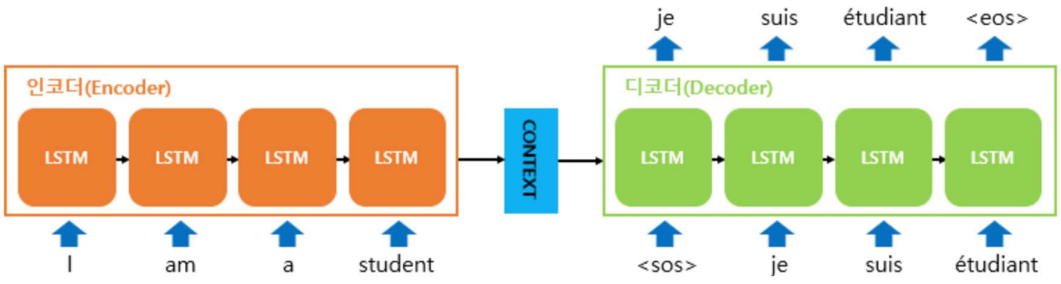


Sequence-to-sequence는 입력된 시퀀스로부터 다른 도메인의 시퀀스를 출력하는 다양한 분야에서 사용되는 모델입니다. 대표적으로 번역기에서 사용되는 모델로 그림처럼 'I am a student'라는 영어 문장을 입력받아 'je suis etudiant'라는 프랑스 문장을 출력할 수 있습니다.



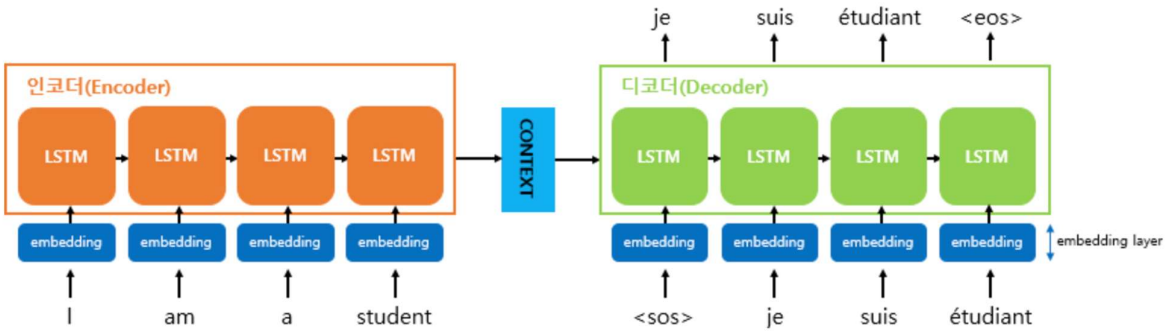
Seq2seq 모델 내부의 모습을 보면, 크게 인코더와 디코더라는 두개의 모듈로 구성됩니다. 인코더는 입력문장의 모든 단어들을 순차적으로 입력받은 뒤에 마지막에 이 모든 단어 정보들을 압축해서 하나의 벡터로 만드는데 이를 컨텍스트 벡터라고 하는데요. 인코더는 이 컨텍스트 벡터를 디코더로 전송하고 디코더는 컨텍스트 벡터를 받아서 번역된 단어를 한 개씩 순차적으로 출력합니다.



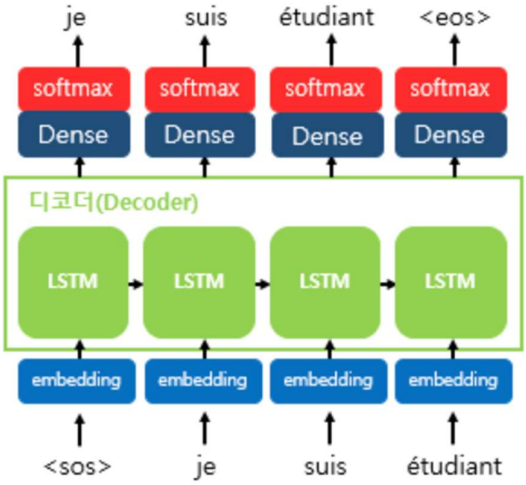
그림에서 주황박스 초록 박스 하나하나는 RNN셀로, 입력 문장을 받는 LSTM 셀을 인코더, 출력 문장을 출력하는 LSTM 셀을 디코더라 합니다. 입력 문장은 단어 단위로 쪼개서 각각 셀의 각 시점에 입력 됩니다. 인코더의 LSTM 셀은 모든 단어를 입력 받은 뒤에 셀의 마지막 시점의 은닉 상태를 디코더로 넘겨주는데 이를 컨텍스트 벡터라고 합니다. 컨텍스틑 벡터는 디코더의 LSTM 셀의 첫번째 은닉상태에서 사용됩니다. 디코더는 초기 입력으로 문장의 시작을 의미하는 <sos>가 들어갑니다. 디코더는 <sos>가 입력되면, 다음에 등장할 확률이 높은 단어를 예측합니다. 첫번째 시점의 디코더 셀은 다음에 등장할 단어로 je를 예측하였습니다. 예측된 단어를 다음 셀의 입력으로

입력합니다. 또, 입력된 단어 je로부터 suis를 예측하고 다시 다음 셀의 입력으로 보냅니다. 디코더는 이런 식으로 다음에 올 단어를 예측하고, 예측된 단어를 다음 셀에 입력으로 넣는 행위를 반복하는데 다음 단어가 <eos>로 예측될 때까지만 반복합니다.

Seq2seq는 훈련과정과 테스트과정의 작동방식이 조금 다릅니다. 테스트 과정에서는 앞서 설명한 과정과 같이 디코더는 오직 컨텍스트 벡터와 <sos>만으로 입력으로 받아 다음에 올 단어를 예측하는데여 이렇게 되면 한번 예측이 틀리게 되면 연쇄작용으로 디코더 전체의 예측과 학습이 비효율적으로 진행될 것입니다. 이런한 문제를 해결하기 위해 훈련과정에서는 교사 강요 학습방법을 적용합니다. 교사 강요란, 학습을 할때 이전 시점의 예측값을 input으로 하지 않고 실제값을 input으로 적용하여 학습하는 방법입니다.



Seq2seq에서 사용되는 모든 단어들은 임베딩 벡터로 변환후 입력으로 사용됩니다. 그림을 보면 각 단어들의 임베딩 층을 볼 수있습니다.



디코더 단어 예측 부분을 보겠습니다. 디코더 셀의 출력으로 단어에 대한 벡터 값이 나올 것입니다. 그 중 확률이 가장 높은 단어를 예측하기 위해 소프트맥스 함수를 사용합니다. 이를 통해 최종 예측 단어를 정합니다.

<seq2seq2 기계번역 훈련>

```
from keras.layers import Input, LSTM

encoder_inputs = Input(shape=(None, src_vocab_size))
encoder_lstm = LSTM(256, return_state=True)

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

인코더를 보면 일반 LSTM모델과 셜계가 동일합니다. 우선 LSTM 은닉 상태 크기는 256으로 선택하고 인코더의 내부상태를 디코더로 넘겨주어야 하기 때문에 return_state=True로 설정합니다.
즉, 인코더에 입력을 넣으면 내부 상태를 리턴합니다. LSTM에서 state_h, state_c를 리턴 받는데 이는 각각 은닉상태와 셀 상태에 해당합니다. LSTM은 은닉상태와 셀 상태 두가지를 가지는데 이 두가지를 encoder_states에 저장하여 디코더에 전달합니다. 이것이 컨텍스트 벡터입니다.

```
from keras.layers import Dense

decoder_inputs = Input(shape=(None, tar_vocab_size))
decoder_lstm = LSTM(256, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs, initial_state=encoder_states)

decoder_softmax_layer = Dense(tar_vocab_size, activation='softmax')
decoder_outputs = decoder_softmax_layer(decoder_outputs)
```

디코더는 인코더의 마지막 은닉상태를 초기은닉상태로 사용합니다.코드에서 initial_state=encoder_states가 이에 해당됩니다. 디코더의 은닉상태 크기도 256으로 줍니다. 디코더도 은닉상태와 셀 상태를 리턴하기는 하지만 훈련 과정에서는 사용하지 않습니다(교사강요). 그 후 Dense layer와 softmax를 통과해 예측 글자에 해당하는 인덱스를 반환합니다.

```
from keras.models import Model

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=128,
          epochs=25,
          validation_split=0.2)
```

앞서 설계한 인코더와 디코더를 결합하여 seq2seq 모델을 설계하여 출력값과 실제값의 오차를 구합니다. 모델의 x값은 인코더 입력과 디코더 입력이 들어가고 y값은 디코더의 실제값을 넣어줍니다. 배치 크기 128 총 25에포크를 학습합니다. 결과를 보면 loss값이 많이 떨어진 것을 볼 수 있습니다.

<seq2seq 기계번역 동작시키기>

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

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

decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]
decoder_outputs, state_h, state_c = decoder_lstm(decoder_inputs, initial_state=decoder_states_inputs)

decoder_states = [state_h, state_c]

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

인코더와 디코더를 설계하는 코드입니다. 인코더는 encoder_inputs와 encoder_states를 사용하는데요 훈련과정에서 정의했던 것들입니다. 디코더는 decoder_lstm의 intital_state=decoder_states_inputs을 해주는데 다음 단어를 예측하기 위해서 초기 상태를 이전 시점의 상태로 사용하는 코드입니다. 훈련 과정에서와는 달리 lstm의 리턴하는 은닉상태를 셀상태를 decoder_states로 받아줍니다.

```
import numpy as np

for seq_index in [0, 10, 100, 1000]:
  input_seq = encoder_input[seq_index: seq_index+1]
  decoded_sentence = predict_decode(input_seq)

  print("입력:", lines.src[seq_index])
  print("정답:", lines.tar[seq_index][1:len(lines.tar[seq_index])-1])
  print("번역:", decoded_sentence[:len(decoded_sentence)-1], '\n')
```

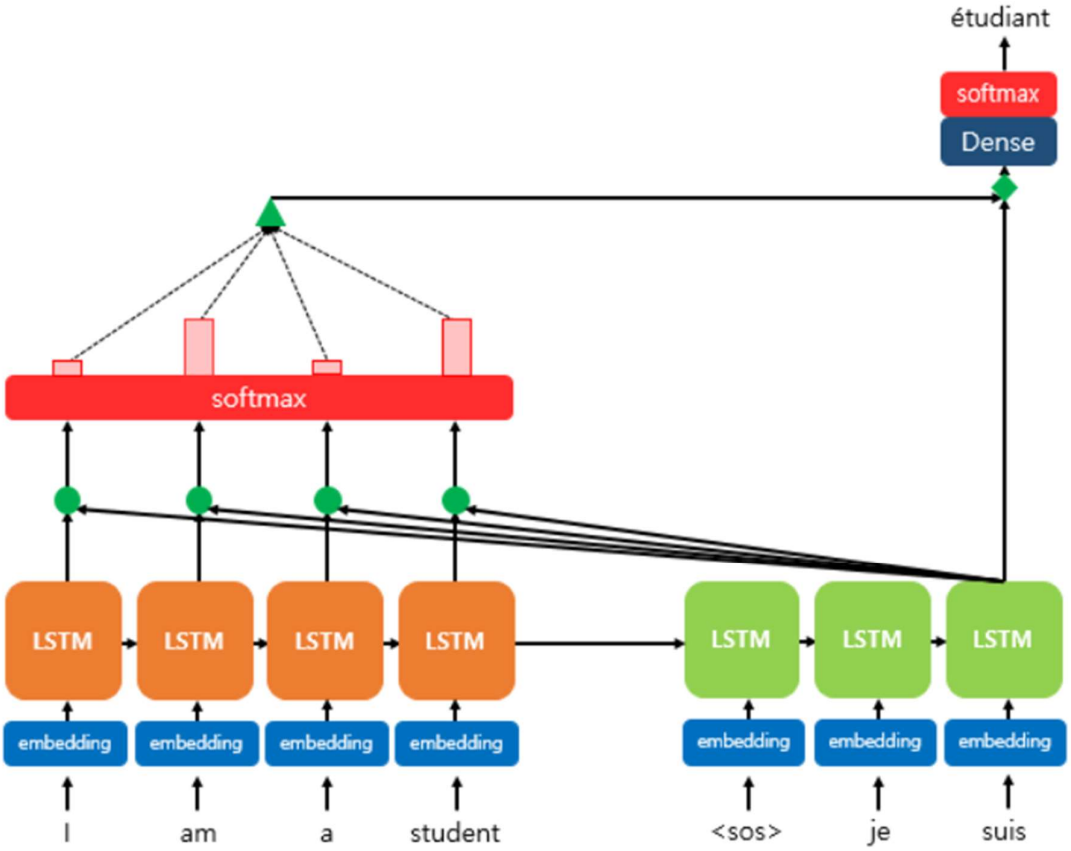
```
입력: Go.
정답:  Va !
번역:  Poursuis.

입력: Stop!
정답:  Ça suffit !
번역:  As-ce de cheveux !

입력: Call us.
정답:  Appelez-nous !
번역:  Appelle-le !

입력: Try it on.
정답:  Essaie-le !
번역:  Essaie encore faim.
```

이런 식으로 출력, 꽤 정확도가 떨어지는 듯

Attention Mechanism

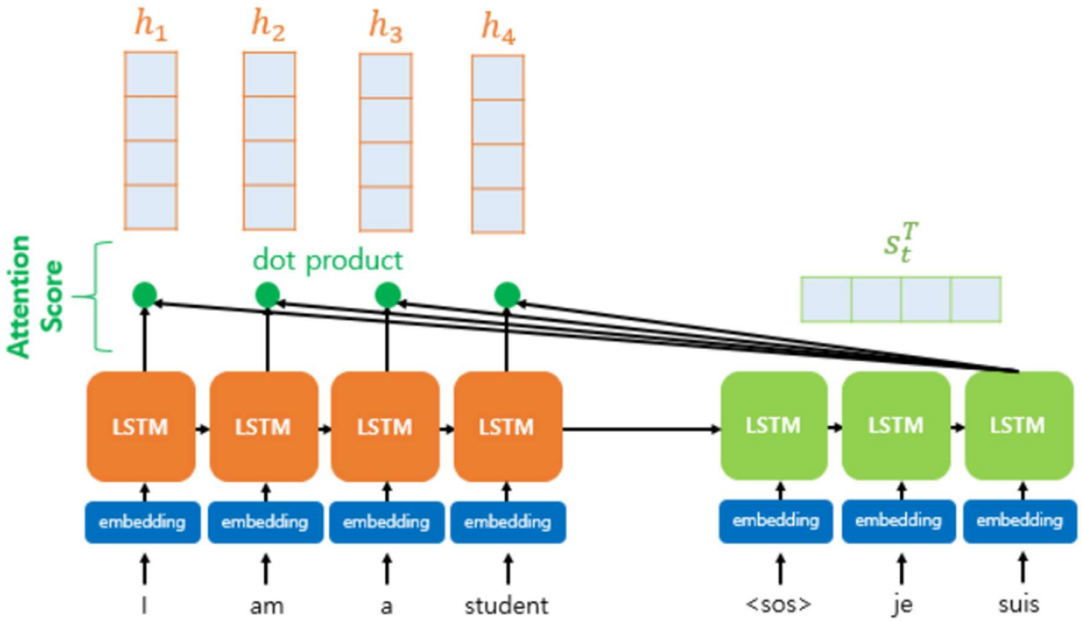
장점: 전체 입력 문장을 단순히 참고하지 않고, 예측할 단어와 연관이 있는 단어를 집중(Attention)해서 참

Attention Mechanism 과정



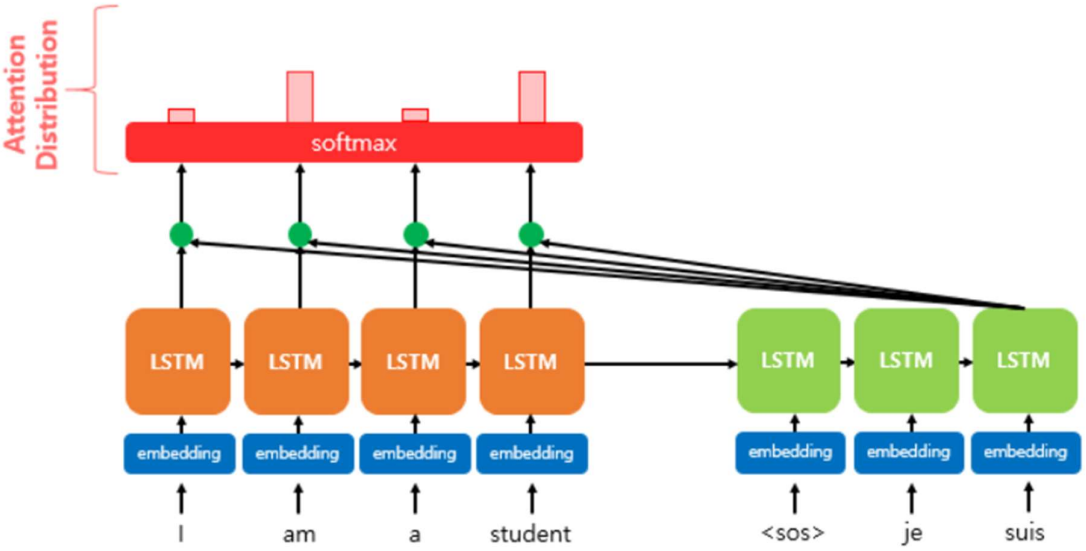
1. attention score 계산
2. 소프트맥스 함수를 통한 attention distribution 계산
3. 각 인코더의 어텐션 가중치와 은닉 상태를 가중합하여 어텐션 값 계산
4. 어텐션 값과 디코더의 t 시범의 은닉 상태를 연결
5. 출력층 연산의 입력이 되는 s~t 계산

attention score 계산



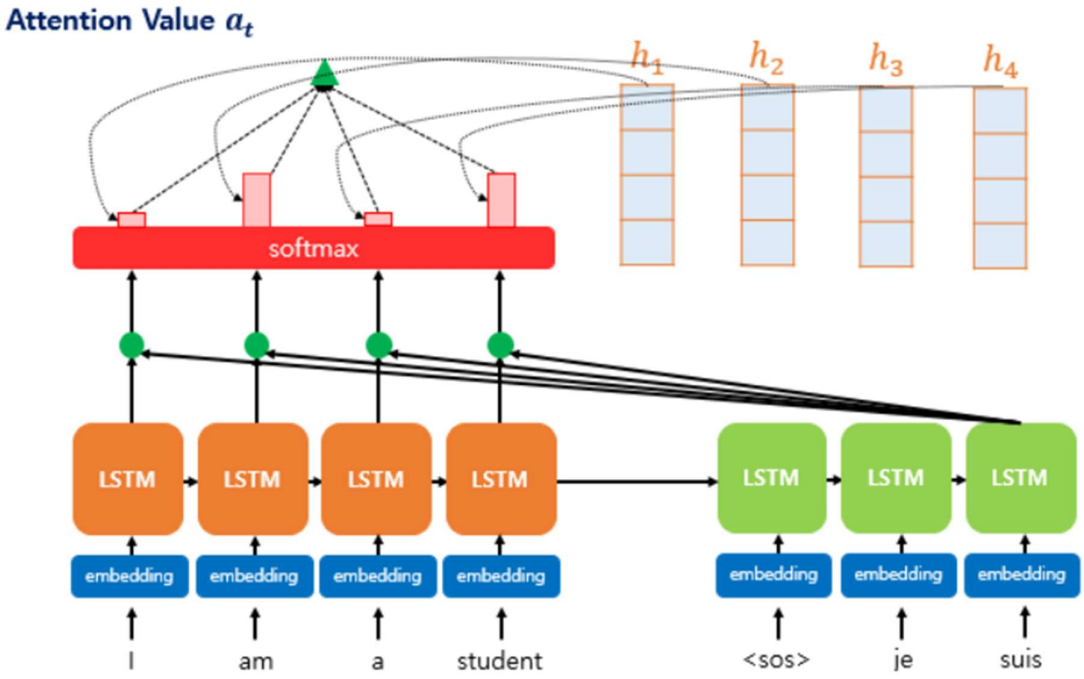
- t 시점 디코더의 은닉 상태를 St 로 지정
- 인코더의 각 시점의 은닉 상태는 h 으로 지정
- dot-product attention 이므로 St 와 h 들을 각각 내적해 각각의 attention score 를 계산

소프트 맥스 함수를 통한 attention distribution 계산



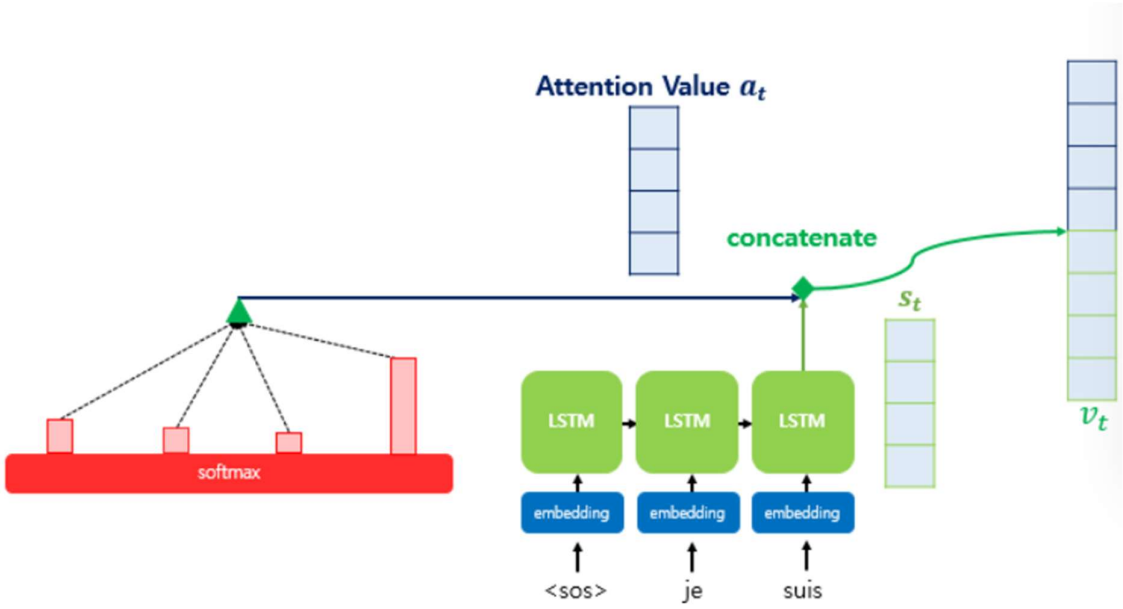
- 각각의 attention score 계산한 값에 softmax 를 적용해 각각의 확률값 계산
- 내적이 크다면 확률값이 높아지고, 작다면 확률값은 작아짐.
- 확률값: 예측할 단어와 연관이 있는 단어를 찾는 것임.

각 인코더의 어텐션 가중치와 은닉 상태를 가중합해 어텐션 값 계산



- 각 인코더의 은닉 상태와 어텐션 가중치들을 곱하고 어텐션 값을 계산

어텐션 값과 디코더의 t 시점의 은닉 상태를 연결



- 어텐션 값을 구했다면 s 와 연결해 예측 연산에 사용할 v 를 계산
- v 는 기존과는 다르게 인코더의 정보를 가지고 있어 좀 더 좋은 성능의 예측을 수행할 수 있음.

인코더(Encoder)

```
encoder_inputs = Input(shape =(None, src_vocab_size))
encoder_lstm = LSTM(256, return_state=True)

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

인코더 과정은 seq2seq 와 동일합니다.

디코더(Decoder)

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

decoder_inputs = Input(shape=(None, tar_vocab_size))
decoder_lstm = LSTM(256, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs, initial_state = encoder_states)

S_ = tf.concat([state_h[:, tf.newaxis, :], decoder_outputs[:, :-1, :]], axis=1)

attention = Attention()
context_vector = attention([S_, encoder_outputs])
concat = tf.concat([decoder_outputs, context_vector], axis=1)
decoder_softmax_layer = Dense(tar_vocab_size, activation='softmax')
decoder_outputs = decoder_softmax_layer(concat)
```

- S_는 은닉 상태와 디코더의 최종 출력을 연결한 결과, 연결할 때 형상을 맞춰주기 위해 축을 추가함

- attention layer 는 디코더의 은닉 상태와 인코더 은닉 상태 전체를 받아 컨텍스트 벡터를 생성함
- 이 때 attention layer 는 앞서 설명한 과정 중 1~3 번째를 수행, 나머지는 사용자가 연결해주어야 함
- 마지막으로 생성한 컨텍스트 벡터와 디코더의 은닉 상태 전체를 이어 softmax layer 에 투입, 인덱스를 예측함

모델 구성 및 학습

```
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=128,
          epochs=25,
          validation_split=0.2)
```

X = [encoder_inputs, decoder_inputs], y= decoder_outputs 를 넣어서 모델링 후 기존 데이터에 학습해봄.

예측

- 예측도 seq2seq 와 동일하나, 추가된 모델 구조를 반영해주어야 함(attention layer)
- 나머지는 seq2seq 와 동일함

```
encoder_model = Model(inputs=encoder_inputs,
                      outputs=[encoder_outputs, encoder_states])
```

```
decoder_state_input_h = Input(shape=(256))
decoder_state_input_c = Input(shape=(256))

estate_h = Input(shape=(256))
encoder_outputs = Input(shape=(256))

decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]
decoder_outputs, state_h, state_c = decoder_lstm(decoder_inputs, initial_state=decoder_states_inputs)

decoder_states = [state_h, state_c]

S_ =tf.concat([estate_h[:, tf.newaxis, :], decoder_outputs[:, :-1, :]], axis=1)
context_vector = attention([S_, encoder_outputs])
decoder_concat = tf.concat([decoder_outputs, context_vector], axis=-1)
decoder_outputs = decoder_softmax_layer(decoder_outputs)
decoder_model = Model(inputs=[decoder_inputs, estate_h, encoder_outputs] + decoder_states_inputs,
                      outputs =[decoder_outputs]+decoder_states)
```

```
idx_to_src = dict((i, char) for char, i in src_to_idx.items())
idx_to_tar = dict((i, char) for char, i in tar_to_idx.items())
```

estate_h 는 디코더에서 인코더의 은닉 상태를 입력해 줍니다.

최종 은닉 상태(encoder_outputs)를 따로 입력받아야 함

```
def predict_decode(input_seq):
  outputs_input, states_value = encoder_model.predict(input_seq)

  target_seq = np.zeros((1,1, tar_vocab_size))
  target_seq[0,0, tar_to_idx['₩t']] = 1

  stop=False
  decoded_sentence = ""

  while not stop:
    output_tokens, h, c = decoder_model.predict([target_seq, states_value[0], outputs_input]+states_value)

    sampled_token_index = np.argmax(output_tokens[0, -1,:])
    sampled_char = idx_to_tar[sampled_token_index]

    decoded_sentence += sampled_char

    if sampled_char == '₩n' or len(decoded_sentence) > max_tar_len:
      stop =  True

    target_seq = np.zeros((1,1, tar_vocab_size))
    target_seq[0, 0, sampled_token_index] = 1.

    states_value = [h,c]

  return decoded_sentence
```

```
import numpy as np

for seq_index in [100, 200, 300, 400]:
  input_seq = encoder_input[seq_index: seq_index+1]
  decoded_sentence = predict_decode(input_seq)

  print("입력:", lines.src[seq_index])
  print("정답:", lines.tar[seq_index][1:len(lines.tar[seq_index])-1])
  print("번역:", decoded_sentence[:len(decoded_sentence) -1], '₩n')
```