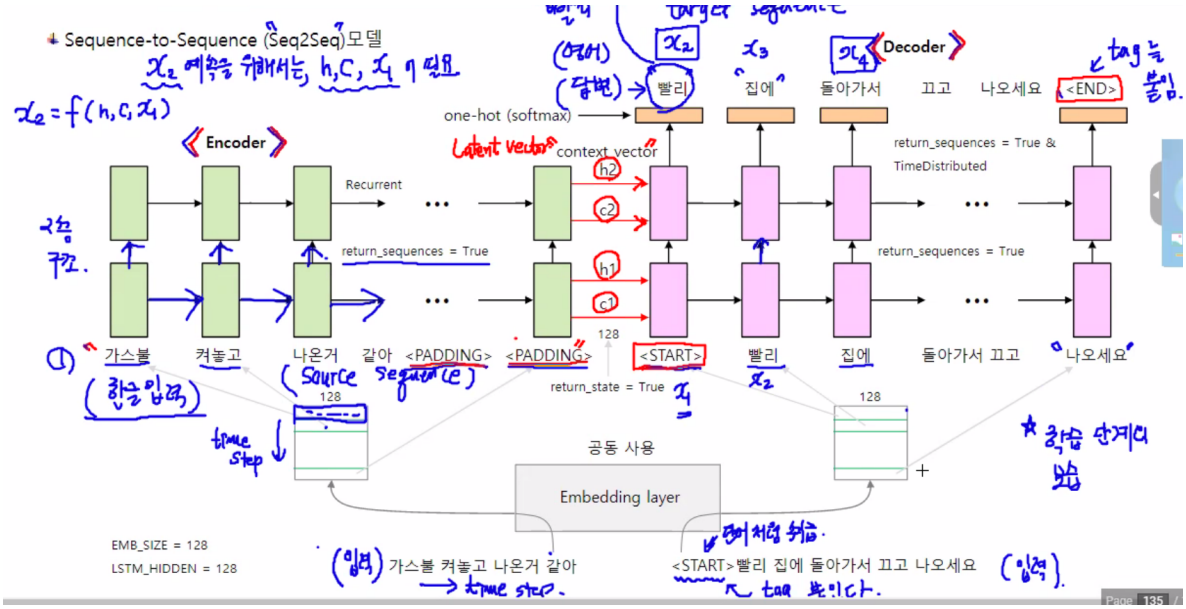
출처가 명시되지 않은 모든 자료(이미지 등)는 조성현 강사님의 강의 기반

# 챗봇(Seq2Seq)

> Seq2Seq 모델을 활용한 챗봇을 만들어 보자.




Seq2Seq 모델의 핵심은 2개의 RNN 네트워크였다. 위의 그림에서와 같이, 인코더와 디코더를 각각 2층의 LSTM 네트워크로 구성하는 챗봇을 만들어 보자.

*Source Sentence*가 `가스불 켜놓고 나온거 같아 <PADDING> <PADDING> ... <PADDING>` 이 되고, *Target Sentence*가 `<START> 빨리 집에 돌아가서 끄고 나오세요 <PADDING> … <END>`가 된다.

인코더에 *Source Sentence*가 입력되었을 때, 디코더에서 *Target Sentence*가 나와야 한다. 이를 위해 학습 모듈과 채팅 모듈을 따로 구성한다. 임베딩 레이어는 공통으로 사용한다.

학습 모듈은 인코더 입력, 디코더 입력이 있으며, 인코더 입력으로 *Source Sentence*가, 과 디코더 입력으로 *Target Sentence*가 들어갈 때 디코더 출력으로 특별 토큰을 제외한 *Target Sentence*가 나오도록 학습한다.

채팅 모듈에는 인코더 입력은 있지만, 디코더 입력이 없다. 따라서 인코더 입력이 이루어질 때 `<START>`부터 시작하여 바로 다음 단어가 나오도록 예측한다.

이론적으로라면 학습 모듈에서도 채팅 모듈에서와 같이 바로 다음 단어를 예측할 수 있도록 학습이 이루어져야 하나, 어차피 학습 모듈에 활용할 데이터를 모두 가지고 있고, 시간 상의 문제가 있기 때문에, 모든 데이터를 한 번에 넣어서 학습한다. 이를 **Teacher Forcing** 방식이라 한다.

# 1. 전처리

원래 데이터의 생김새는 다음과 같다.

| | Q | A | label |
|---|---|---|---|
| 0 | 12시 땡! | 하루가 또 가네요. | 0 |
| 1 | 1지망 학교 떨어졌어 | 위로해 드립니다. | 0 |
| 2 | 3박4일 놀러가고 싶다 | 여행은 언제나 좋죠. | 0 |
| 3 | 3박4일 정도 놀러가고 싶다 | 여행은 언제나 좋죠. | 0 |
| 4 | PPL 심하네 | 눈살이 찌푸려지죠. | 0 |
| ... | ... | ... | ... |
| 11818 | 훔쳐보는 것도 눈치 보임. | 티가 나니까 눈치가 보이는 거죠! | 2 |
| 11819 | 훔쳐보는 것도 눈치 보임. | 훔쳐보는 거 티나나봐요. | 2 |
| 11820 | 흑기사 해주는 짝남. | 설렜겠어요. | 2 |
| 11821 | 힘든 연애 좋은 연애라는게 무슨 차이일까? | 잘 헤어질 수 있는 사이 여부인 거 같아요. | 2 |
| 11822 | 힘들어서 결혼할까봐 | 도피성 결혼은 하지 않길 바라요. | 2 |

Q는 질문으로서 인코더에 들어갈 입력을, A는 답변으로서 디코더의 입력과 출력이 된다. 라벨 0, 1, 2는 각각 인코더의 입력, 디코더의 입력, 디코더의 타겟을 나타낸다.

라벨과 필터링할 특수 문자들, 패딩, 시작과 끝 등의 특수 토큰을 미리 파라미터로 정의해 놓는다. 특히 필터링할 특수 문자(구둣점)의 경우, `re.compile` 함수를 이용해 미리 정규표현식 형태로 컴파일해 둔다.

또한, 원래는 위의 함수로 형태소 분석을 진행해야 하지만, 답변을 위해 형태소로 대답할 수 없으므로, 형태소 분석은 진행하지 않는다.

```
TOKENIZE_AS_MORPH = False       # 형태소 분석 여부
ENC_INPUT = 0                   # encoder 입력을 의미함
DEC_INPUT = 1                   # decoder 입력을 의미함
DEC_TARGET = 2                  # decoder 출력을 의미함
MAX_SEQUENCE_LEN = 10           # 단어 시퀀스 길이

FILTERS = "([~.,!?\"':;)(])"
PAD = "<PADDING>"
STD = "<START>"
END = "<END>"
UNK = "<UNKNOWN>"

MARKER = [PAD, STD, END, UNK]
CHANGE_FILTER = re.compile(FILTERS)
```

- 데이터 로드 : 로드 단계에서 바로 학습과 평가 세트로 나눈다.

```
def load_data():
    data_df = pd.read_csv(DATA_PATH, header=0)
    question, answer = list(data_df['Q']), list(data_df['A'])

    train_input, eval_input, train_label, eval_label = \
        train_test_split(question, answer, test_size=0.1, random_state=42)

    return train_input, train_label, eval_input, eval_label
```

- 인코더, 디코더의 입력과 출력 데이터

인코더의 입력은 `Q`에 해당하는 게 되고, 디코더의 입력은 `A`에 해당하는 게 된다. `A`에 해당하는 문장의 앞 뒤에는 특수 토큰(각각 `<START>`와 `<END>`)을 붙여 준다. 모델은 인코더와 디코더에 각각 문장이 입력되었을 때, 디코더에서 특수 토큰을 제외한 문장이 나오도록 학습한다. 예컨대, 아래와 같은 방식이다.

> - 인코더 입력 : "가끔 궁금해" -> [9310, 17707, 0, 0, 0, 0, 0, 0, 0, 0]
> - 디코더 입력 : "그 사람도 그럴 거예요" -> [STD, 20190, 4221, 13697, 14552, 0, ...]
> - 디코더 타켓 : [20190, 4221, 13697, 14552, END, 0, ...]

이를 코드로 구현하면 다음과 같다.

```
def data_processing(value, dictionary, pType):
    # 형태소 토크나이징 사용 유무
    if TOKENIZE_AS_MORPH:
        value = prepro_like_morphlized(value)

    sequences_input_index = []
    for sequence in value:
        sequence = re.sub(CHANGE_FILTER, "", sequence)

        if pType == DEC_INPUT:
            # 디코더 입력: <START>로 시작.
            sequence_index = [dictionary[STD]]
        else:
            sequence_index = []

        for word in sequence.split():
            # OOV : <UNK> 토큰.
            if dictionary.get(word) is not None:
                sequence_index.append(dictionary[word])
            else:
                sequence_index.append(dictionary[UNK])

            # 문장 최대 길이 제한
            if len(sequence_index) >= MAX_SEQUENCE_LEN:
                break
```

```
        # 디코더 출력: <END>로 끝.
        if pType == DEC_TARGET:
            if len(sequence_index) < MAX_SEQUENCE_LEN:
                sequence_index.append(dictionary[END])
            else:
                sequence_index[len(sequence_index)-1] = dictionary[END]

        # 패딩
        sequence_index += (MAX_SEQUENCE_LEN - len(sequence_index)) *
[dictionary[PAD]]
        sequences_input_index.append(sequence_index)

    return np.asarray(sequences_input_index)
```

이후 토크나이징하고, 어휘 사전을 만든다. 이전과 같은 방식이다. 이후 전처리된 데이터를 미리 저장해 둔다.

## 2. 학습 모듈

`1`에서 만들어 놓은 vocabulary 파일, 학습 데이터와 평가 데이터를 미리 읽어 온다. 한편, `LOAD_MODEL` 변수를 활용해 추가 학습 여부를 선택한다.

임베딩은 인코더, 디코더 모두 공통으로 사용한다.

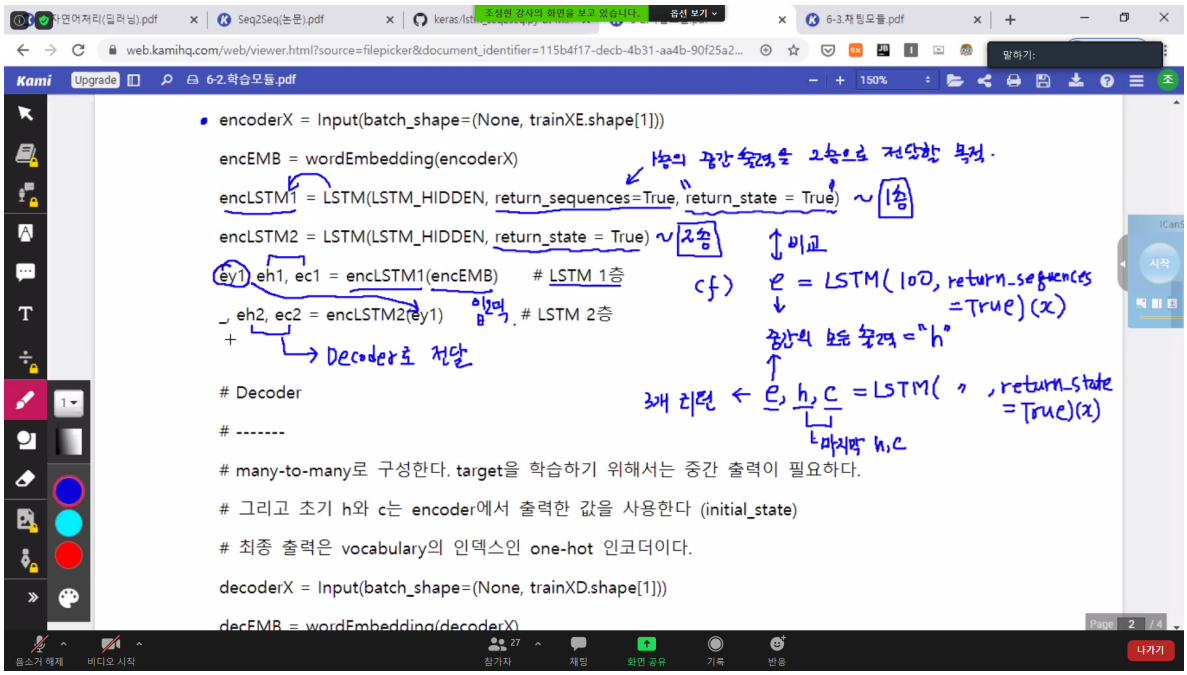
```
VOCAB_SIZE = len(idx2word)
EMB_SIZE = 128
LSTM_HIDDEN = 128
MODEL_PATH = './dataset/6-2.Seq2Seq.h5'
LOAD_MODEL = False

# 워드 임베딩
K.clear_session()
wordEmbedding = Embedding(input_dim=VOCAB_SIZE, output_dim=EMB_SIZE)
```

- 인코더 네트워크 구성

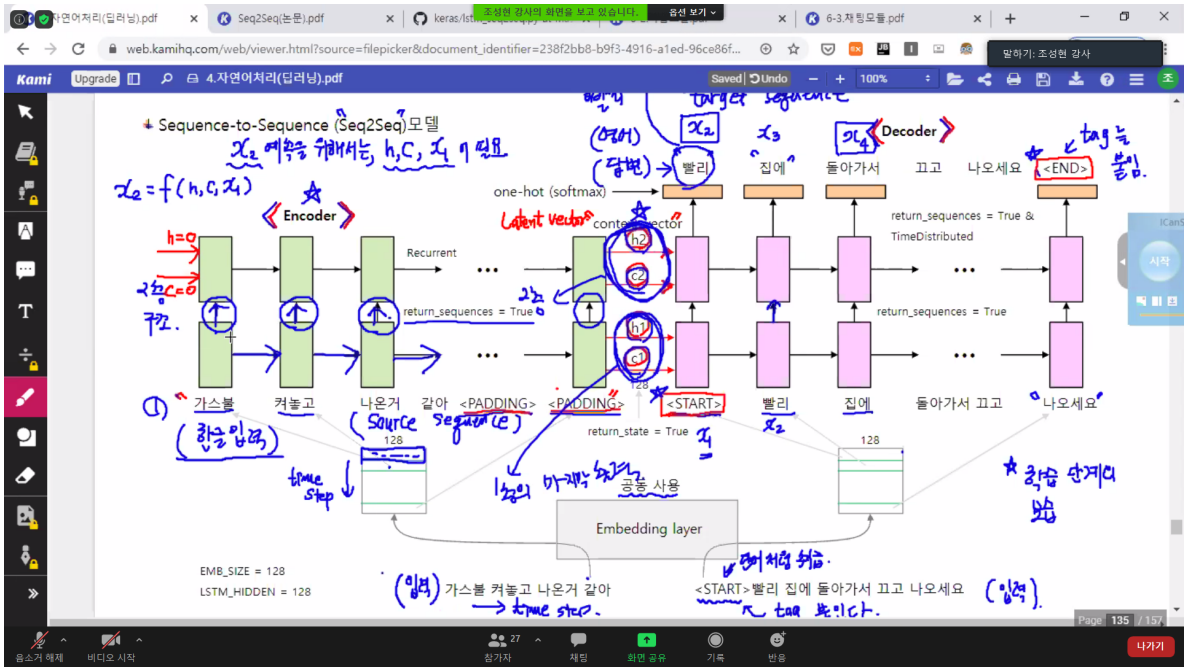
`many-to-one` 방식으로 구성한다. 중간 출력은 필요하지 않고, decoder로 전달할 최종 출력인 `h`와 `c`만 필요하다. 최종 `h`와 `c`는 이전 문맥의 정보를 모두 갖고 있는 latent feature이자 context vector가 된다. 이를 얻기 위해 `return_state=True`로 설정한다.

```
# 인코더
encoderX = Input(batch_shape=(None, trainXE.shape[1]))
encEMB = wordEmbedding(encoderX)
encLSTM1 = LSTM(LSTM_HIDDEN, return_sequences=True, return_state = True)
encLSTM2 = LSTM(LSTM_HIDDEN, return_state = True)
ey1, eh1, ec1 = encLSTM1(encEMB)     # LSTM 1층
_, eh2, ec2 = encLSTM2(ey1)          # LSTM 2층
```




return_sequences 옵션 줘서 ey1이 전달되고, eh1, ec1 ㅇ,은 출력.

> *참고* : `return_sequences` vs `return_state`
>
> `return_sequences=True` 옵션을 주면 중간 출력을 모두 쌓아 놓고 전달'만' 한다.
> `reutrn_state=True` 옵션을 주면 중간의 모든 hidden state 출력과 최종 출력 h, c를 반환한다.
>
> 마지막 부분에서 나오는 최종 h, c를 디코더 네트워크로 전달만 한다.
>
> ```
> e = LSTM(100, return_sequences=True)(x) # 중간 출력 "전달"
> e, h, c = LSTM(100, return_sequences=True, return_state=True)(x) # 중간 출력, 최종 출력 "반환"
> ```
>
> 한편, `return_state' 파라미터를 True로 설정하고 인덱싱을 사용할 수도 있다.
>
> ```
> lstm = LSTM(50, return_sequences = True, return_state = True)(Input(shape = (10, 30)))
> print(lstm[0].shape)          # output : (?, ?, 50)
> print(lstm[1].shape)          # hidden state : (?, 50)
> print(lstm[2].shape)          # cell state: (?, 50)
> ```




1층 LSTM에서 `return_sequences` 옵션을 준 것은 그 다음 층으로 출력을 전달하기 위함이다. 또한, 디코더로 전달할 마지막 h, c 값(context vector)도 필요하기 때문에, `return_state` 옵션을 준다. 2층에서는 `return_sequences` 옵션으로 출력을 전달할 필요가 없기 때문에 해당 옵션을 주지 않는다.

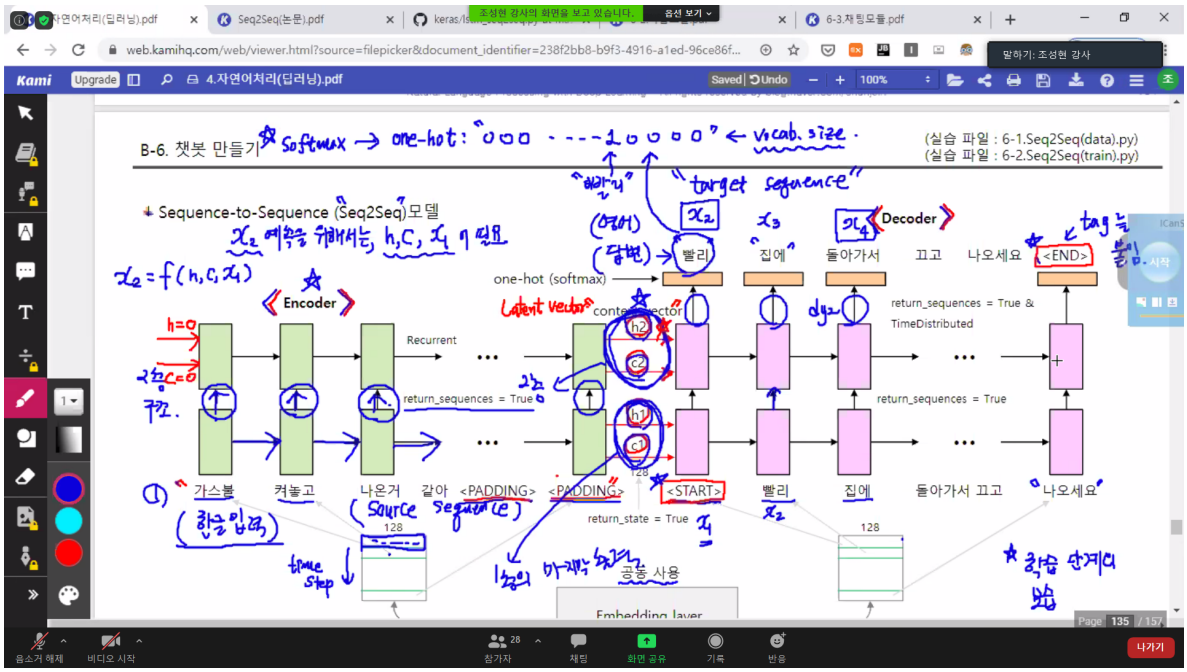
- 디코더 네트워크 구성

`many-to-many` 방식으로 구성한다. 디코더 입력을 input으로 받고, `<START>`로 시작해서 대답 문장이 나오는 구조이다.

```
# 디코더
decoderX = Input(batch_shape=(None, trainXD.shape[1]))
decEMB = wordEmbedding(decoderX)
decLSTM1 = LSTM(LSTM_HIDDEN, return_sequences=True, return_state=True)
decLSTM2 = LSTM(LSTM_HIDDEN, return_sequences=True, return_state=True)
dy1, _, _ = decLSTM1(decEMB, initial_state = [eh1, ec1])
dy2, _, _ = decLSTM2(dy1, initial_state = [eh2, ec2])
decOutput = TimeDistributed(Dense(VOCAB_SIZE, activation='softmax'))
outputY = decOutput(dy2)
```

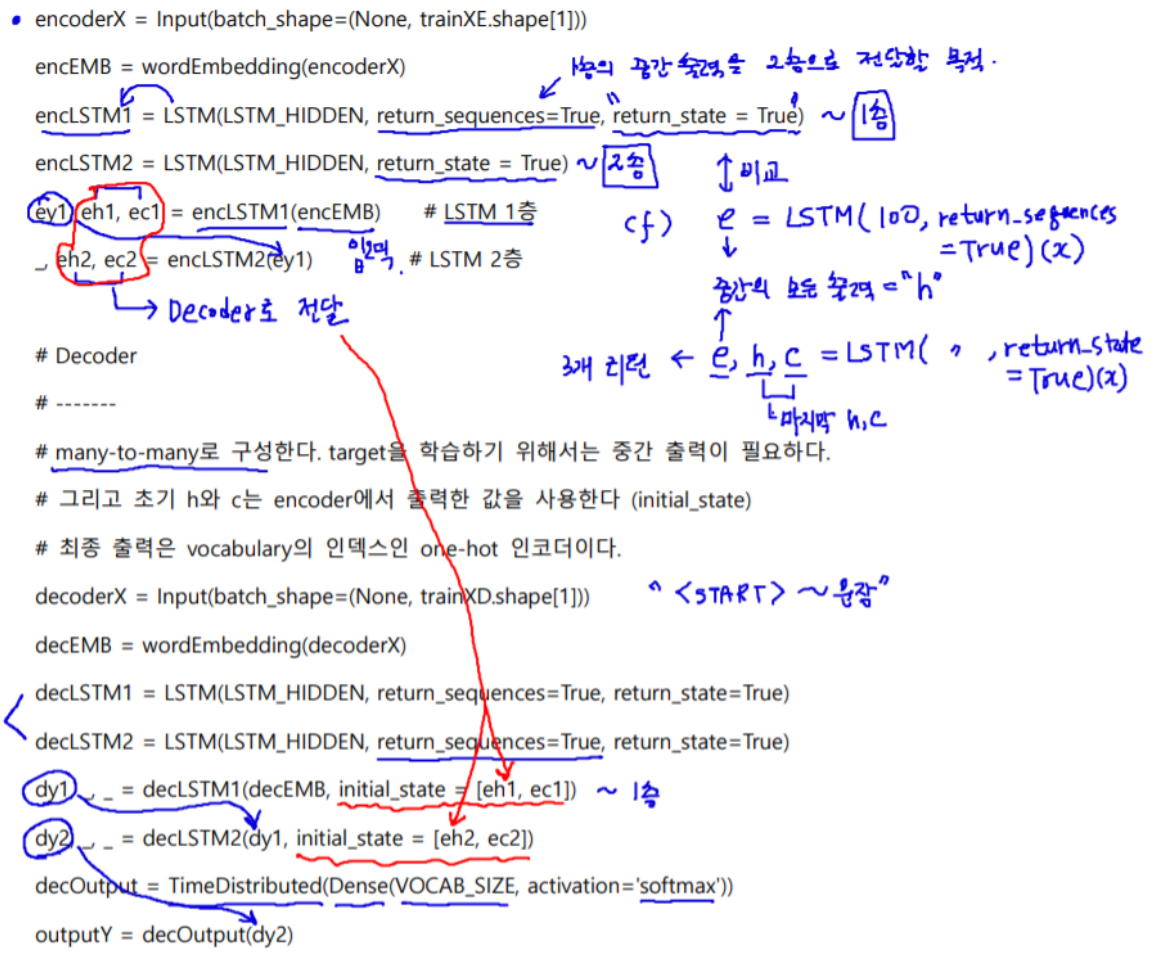
두 층 모두 `many-to-many`이므로, 두 층 모두에 `return_sequences=True` 옵션을 준다. 특징적인 것은, 인코더 문장에서 전달되어 온 context vector `h`와 `c`가 초기값(`initial_state`)으로 설정된다는 것이다. 아래 코드로 보면, `eh1`, `ec1`이 인코더에서 디코더로 전달된다.

특히 디코더에서는 최종 출력이 필요하다. 따라서 1층의 전체 출력 `dy1`은 2층에 전달될 목적으로, 2층의 `dy2`는 실제 라벨 문장을 예측할 용도로 사용한다. 최종 출력은 vocabulary의 인덱스에 1이 들어 있는 원핫 벡터가 된다.




한편, `many-to-many`이므로 각각의 타임 스텝마다 오류를 분배하여 역전파하기 위해, `TimeDistributed` 함수를 사용한다.

코드 상에서 가중치가 전달되는 그림을 보면 다음과 같다.



- 전체 모델 구성

인코더와 디코더 네트워크를 연결해 전체 모델을 구성한다. input으로 인코더 입력과 디코더 입력이 들어 가고, 디코더 출력이 output으로 나온다. 디코더 출력을 범주형 변수로 바꾸지 않았기 때문에, 원핫 벡터의 상태이다. 따라서 컴파일 시 loss를 `sparse_categorical_crossentropy`로 설정한다.

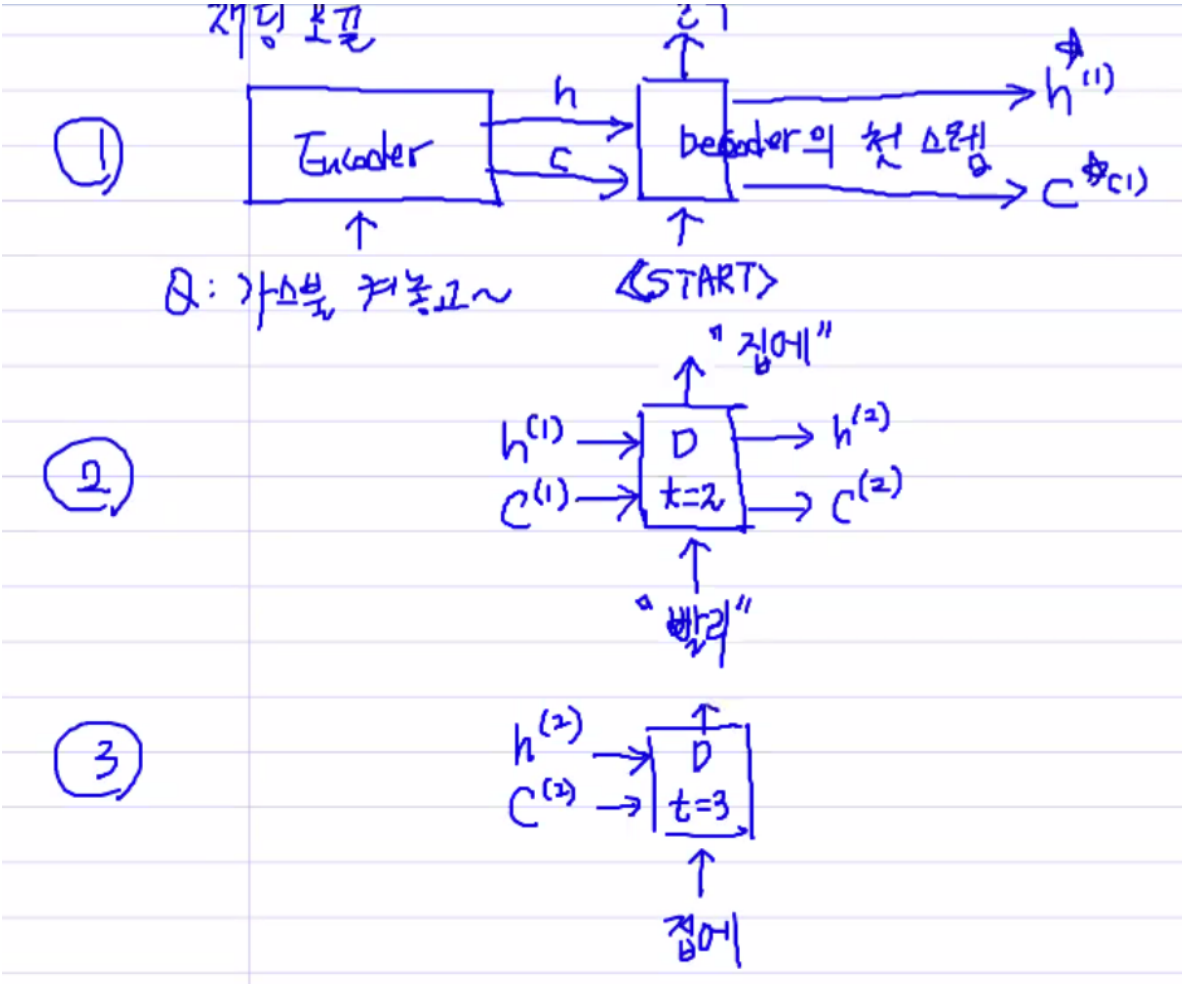
> *참고*
>
> 만약 `categorical_crossentropy`를 사용하고 싶다면, 디코더 출력을 `to_categorical`을 사용해 범주형으로 바꾸면 된다. 그러나 그렇게 모델을 구성할 경우, 일일이 범주형으로 바꾸어 그 다음 출력을 내놓기 때문에 시간이 오래 걸린다. 챗봇 대답에 사용하기에는 적절하지 않다.

# 3. 채팅 모듈

학습된 모듈을 바탕으로 채팅을 한다. 어떻게 보면 **예측**이다. 상술했듯, **Teacher Forcing** 방식으로 학습이 이루어졌지만, 예측 시에는 인코더의 입력만 존재한다. 따라서 다음과 같은 방식으로 각 스텝마다 예측을 반복해 나간다.

- Encoder 입력 질문을 input에 넣는다. `h`, `c` 값을 알아낼 수 있다.
- Decoder 첫 번째 스텝에 `h`, `c`와 `<START>` 토큰을 넣어 가동시킨다.
- 두 번째 스텝에 첫 번째 스텝의 `h`, `c`와 출력값을 넣어 가동시킨다.

그림으로 나타내면 다음과 같다.




그런데 timestep이 1이다. 따라서 LSTM 모듈이 알아서 recurrent step을 작동시킬 수 없다. 직접 반복문을 사용해 recurrent step을 돌도록 설정한다. 예전에 LSTM 모델링을 통해 예측할 때 예측된 값을 하나씩 뒤에 붙여서 새로 예측했던 것과 같이, 예측 값을 업데이트해준다.

- 학습 모듈 모델 구조를 그대로 가져 온다.

이미 학습된 모델의 가중치를 업데이트한다. 달라지는 부분은 디코더 입력 부분이다. 원래는 디코더 입력으로 모든 문장을 한 번에 받았지만, 이번에는 토큰 1개씩 받기 때문에, batch shape을 수정한다.

time step이 1이지만, 네트워크 전체의 파라미터는 동일하다.

```
# 워드 임베딩
K.clear_session()
wordEmbedding = Embedding(input_dim=VOCAB_SIZE, output_dim=EMB_SIZE)

# Encoder
encoderX = Input(batch_shape=(None, MAX_SEQUENCE_LEN))
encEMB = wordEmbedding(encoderX)
encLSTM1 = LSTM(LSTM_HIDDEN, return_sequences=True, return_state = True)
encLSTM2 = LSTM(LSTM_HIDDEN, return_state = True)
ey1, eh1, ec1 = encLSTM1(encEMB)    # LSTM 1층
_, eh2, ec2 = encLSTM2(ey1)         # LSTM 2층

# Decoder
decoderX = Input(batch_shape=(None, 1))
decEMB = wordEmbedding(decoderX)
decLSTM1 = LSTM(LSTM_HIDDEN, return_sequences=True, return_state=True)
```

```
decLSTM2 = LSTM(LSTM_HIDDEN, return_sequences=True, return_state=True)
dy1, _, _ = decLSTM1(decEMB, initial_state = [eh1, ec1])
dy2, _, _ = decLSTM2(dy1, initial_state = [eh2, ec2])
decOutput = TimeDistributed(Dense(VOCAB_SIZE, activation='softmax'))
outputY = decOutput(dy2)

# Model
model = Model([encoderX, decoderX], outputY)
model.load_weights(MODEL_PATH)
```

마지막에 모델을 구성할 때, 미리 학습시켜 놓은 가중치를 업데이트하는 것을 잊지 말자.

- 채팅용 모델

입력으로 들어가야 할 것에 각 LSTM 인코더 층의 초기 `h`, `c` 파라미터를 설정해  주어야 한다. 그리고 인코더 입력이 들어 가면 인코더 출력으로 `h`, `c`까지 출력이 4개가 나오도록 한다.

이렇게 나온 출력을 디코더 LSTM 1층, 디코더 LSTM 2에 각각의 `initial_state`로 넣어 준다.

`ih1`, `ic1`, `ih2`, `ic2`가 한 스텝짜리 LSTM으로 들어 가면, `dec_OUTPUT`이 출력으로 나오게 학습한다.

```
# Chatting model
model_enc = Model(encoderX, [eh1, ec1, eh2, ec2])

ih1 = Input(batch_shape = (None, LSTM_HIDDEN))
ic1 = Input(batch_shape = (None, LSTM_HIDDEN))
ih2 = Input(batch_shape = (None, LSTM_HIDDEN))
ic2 = Input(batch_shape = (None, LSTM_HIDDEN))

dec_output1, dh1, dc1 = decLSTM1(decEMB, initial_state = [ih1, ic1])
dec_output2, dh2, dc2 = decLSTM2(dec_output1, initial_state = [ih2, ic2])

dec_output = decOutput(dec_output2)
model_dec = Model([decoderX, ih1, ic1, ih2, ic2],
                  [dec_output, dh1, dc1, dh2, dc2])
```

- Question을 입력 받아 Answer를 생성하는 함수

인코더 모델로부터 question을 입력 받아 초기 `h`, `c`를 생성한다. 그리고 시작 단어를 `word2idx`에서 `<START>`로 찾는다. 이후, 최대 문장 길이까지 디코더 모델이 단어와 `h`, `c`를 입력 받아 예측하도록 한다.

`argmax`를 통해 다음 단어의 위치를 찾고 해당 단어를 채택한다. 디코더의 다음 단어를 추가하고, `h`, `c`의 파라미터를 업데이트하는 과정을 거친다.

만약 예상한 단어가 `<END>`이거나 `<PADDING>`이라면 더 이상 예상할 게 없으므로 반복문 순회를 종료한다.

```
# Question에 대한 Answer 생성
def genAnswer(question):
    question = question[np.newaxis, :]
    init_h1, init_c1, init_h2, init_c2 = model_enc.predict(question)

    # 시작 단어: <START>
    word = np.array(word2idx['<START>']).reshape(1, 1)

    answer = []
    for i in range(MAX_SEQUENCE_LEN):
        dY, next_h1, next_c1, next_h2, next_c2 = \
            model_dec.predict([word, init_h1, init_c1, init_h2, init_c2])

        # 디코더의 출력: vocabulary에서 argmax
        nextWord = np.argmax(dY[0, 0])

        # 종료 조건
        if nextWord == word2idx['<END>'] or nextWord == word2idx['<PADDING>']:
            break

        # 예상 단어 출력 추가
        answer.append(idx2word[nextWord])

        # 다음 recurrent를 위해 h, c 업데이트
        word = np.array(nextWord).reshape(1,1)

        init_h1 = next_h1
        init_c1 = next_c1
        init_h2 = next_h2
        init_c2 = next_c2

    return ' '.join(answer)
```

- 챗봇과 대화하는 함수

질문을 토큰 단위로 분리하고, OOV 및 패딩 처리를 한다. `getAnswer` 함수를 사용해 답을 얻어 온다.

```
# Chatting
def chatting(n=100):
    for i in range(n):
        question = input('Q : ')

        if  question == 'quit':
            break

        q_idx = []
        for x in question.split(' '):
            if x in word2idx:
                q_idx.append(word2idx[x])
            else: # OOV
                q_idx.append(word2idx['<UNKNOWN>'])
```

```
        # 패딩
        if len(q_idx) < MAX_SEQUENCE_LEN:
            q_idx.extend([word2idx['<PADDING>']] * (MAX_SEQUENCE_LEN -
len(q_idx)))
        else:
            q_idx = q_idx[0:MAX_SEQUENCE_LEN]

        answer = getAnswer(np.array(q_idx))
        print('A :', answer)
```