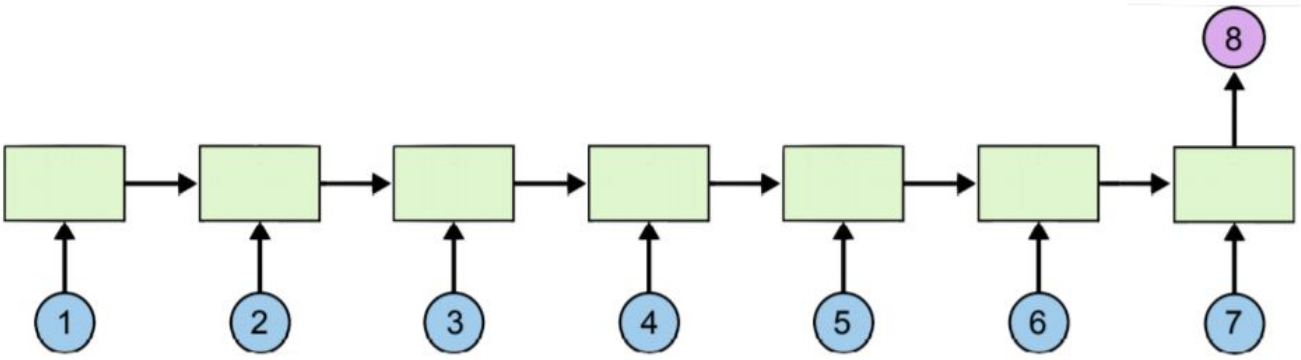
2020 SUMMER ESC :: Week 6 RNN

LAB 11-4 RNN Time Series

- 시계열 분석?

시계열 분석은 관측지 $x_1, x_2 \ldots x_n$가 시간적 순서를 가진 시계열 데이터에서 과거의 값을 통해 미래의 값을 예측하는 것이다. 시계열 분석을 위해 RNN 모델을 사용하고자 한다. 아래의 예시를 보자



Day	1	2	3	4	5	6	7
Open	828.659973	823.02002	823.02002	819.929993	819	816	811.700012
High	833.450012	828.070007	824.400024	823	823	820.958984	815.25
Low	828.349976	821.655029	818.97998	818.469971	816	815.48999	809.780029
Volume	1247700	1597800	1281700	1304000	1053600	1198100	1129100
Close	831.659973	828.070007	824.159973	818.97998	820.450012	819.23999	813.669983

이 데이터는 일별 구글 주가데이터로, 시가 최고가 최저가 거래량 종가를 입력해 8일차의 종가를 예측하고자 한다. 이 모델은 8일차의 종가를 예측하기 위해서 그 전 일주일 치의 데이터를 보면 된다! 라는 가정을 하고 있다(실제와는 다른 가정..) 우리가 알고 있는 일반적인 시계열 분석의 경우 y변수를 trend, seasonality, noise로 나누어 예측을 하는 반면, RNN은 y변수 외에도 여러 feature들을 입력받아 값을 예측한다는 점에서 차이가 있다

코드를 살펴보자!

In [1]:
```
import torch
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
xy = np.loadtxt("data-02-stock_daily.csv", delimiter=",")
xy[0]
```

Out[2]:
```
array([8.28659973e+02, 8.33450012e+02, 8.28349976e+02, 1.24770000e+06,
       8.31659973e+02])
```

- input data의 차원은 5
- 일주일 치의 데이터를 사용할 것이기 때문에 sequence의 길이는 7
- output dimension은 종가를 예측하는 것이기 때문에 1
- hidden layer의 경우 10으로 지정하고 있는데, input data를 받아서 하나의 값으로 압축을 해 다음 cell에 전달을 해주는 것은 모델에게 부담을 준다. 따라서 hidden layer의 차원을 충분히 보장해 주는 것

In [3]:
```
seq_length = 7
data_dim = 5
hidden_dim = 10
output_dim = 1
learning_rate = 0.01
iterations = 500
```

In [4]:
```
# load data
xy = np.loadtxt("data-02-stock_daily.csv", delimiter=",")
xy = xy[::-1]  # reverse order

# split train-test set
train_size = int(len(xy) * 0.7)
train_set = xy[0:train_size]
test_set = xy[train_size - seq_length:]
```

In [5]:
```
def minmax_scaler(data):
    numerator = data - np.min(data, 0)
    denominator = np.max(data, 0) - np.min(data, 0)
    return numerator / (denominator + 1e-7)
```

In [6]:
```
train_set = minmax_scaler(train_set)
test_set = minmax_scaler(test_set)
```

- 각 feature의 값의 단위가 다르기 때문에 min-max scaling을 해준다. 학습에서 부담을 덜어줄 수 있다

In [7]:
```
def build_dataset(time_series, seq_length):
    dataX = []
    dataY = []
    for i in range(0, len(time_series) - seq_length):
        _x = time_series[i:i + seq_length, :] #7일치 데이터
        _y = time_series[i + seq_length, [-1]] #그 다음날의 종가
        dataX.append(_x)
        dataY.append(_y)
    return np.array(dataX), np.array(dataY)
```

In [8]:
```
trainX, trainY = build_dataset(train_set, seq_length)
testX, testY = build_dataset(test_set, seq_length)
```

In [9]:
```
print(trainX[0])
```

```
[[2.53065030e-01 2.45070970e-01 2.33983036e-01 4.66075110e-04
  2.32039560e-01]
 [2.29604366e-01 2.39728936e-01 2.54567513e-01 2.98467330e-03
  2.37426028e-01]
 [2.49235510e-01 2.41668371e-01 2.48338489e-01 2.59926504e-04
  2.26793794e-01]
 [2.21013495e-01 2.46602231e-01 2.54710584e-01 0.00000000e+00
  2.62668239e-01]
 [3.63433786e-01 3.70389871e-01 2.67168847e-01 1.24764722e-02
  2.62105010e-01]
 [2.59447633e-01 3.10673724e-01 2.74113889e-01 4.56323384e-01
  2.71751265e-01]
 [2.76008150e-01 2.78314566e-01 1.98470380e-01 5.70171193e-01
  1.78104644e-01]]
```

In [10]:
```
print(trainY[0])
```

```
[0.16053716]
```

In [11]:
```
# convert to tensor
trainX_tensor = torch.FloatTensor(trainX)
trainY_tensor = torch.FloatTensor(trainY)

testX_tensor = torch.FloatTensor(testX)
testY_tensor = torch.FloatTensor(testY)
```

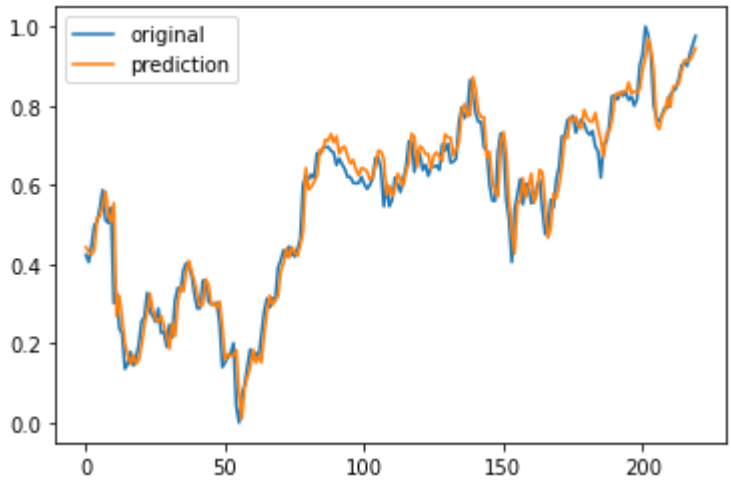
In [12]:
```
class Net(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, layers):
        super(Net, self).__init__()
        self.rnn = torch.nn.LSTM(input_dim, hidden_dim, num_layers=layers, batch_first=Tru
e)
        self.fc = torch.nn.Linear(hidden_dim, output_dim, bias=True)

    def forward(self, x):
        x, _status = self.rnn(x)
        x = self.fc(x[:, -1]) # OUTPUT에서 마지막 것만 사용하겠다. 이 값을 FC에 넣어서 예측
값을 구한다
        return x
```

In [13]:
```
net = Net(data_dim, hidden_dim, output_dim, 1)
```

In [14]:
```
# loss & optimizer setting
criterion = torch.nn.MSELoss()
optimizer = optim.Adam(net.parameters(), lr=learning_rate)
```

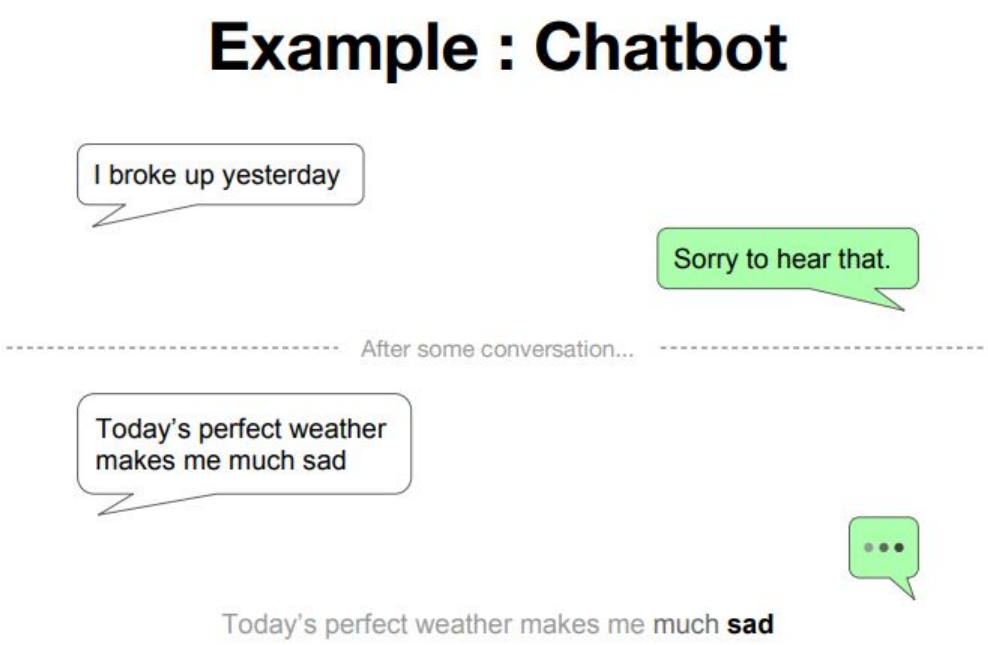
In [15]:
```
# start training
for i in range(iterations):

    optimizer.zero_grad()
    outputs = net(trainX_tensor)
    loss = criterion(outputs, trainY_tensor)
    loss.backward()
    optimizer.step()
```

In [16]:
```
plt.plot(testY)
plt.plot(net(testX_tensor).data.numpy())
plt.legend(['original', 'prediction'])
plt.show()
```



11.5 Sequence-to-Sequence

- 번역기, 챗봇에서 사용

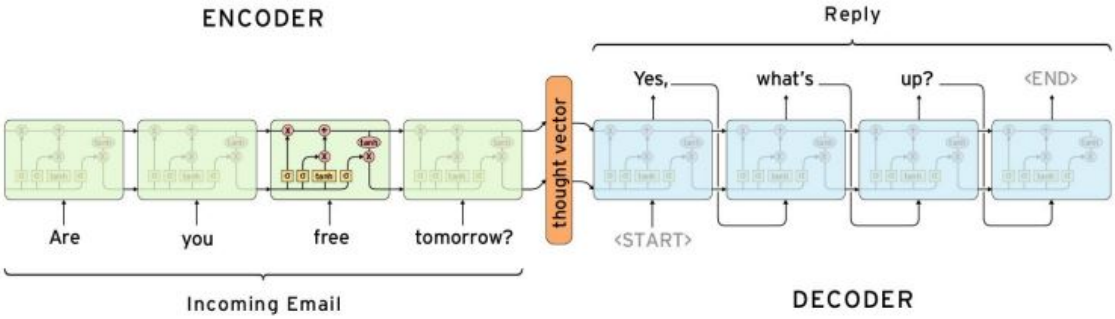


=> 문장이라는 sequential data를 입력 받았을 때 말을 중간까지만 듣고 대답을 할 경우 잘못된 대답을 할 확률이 높아 '말을 끝까지 듣고 말하기'라는 맥락에서 나온 모델

Encoder-Decoder

- seq2seq의 대표 특징. RNN 2개 생성한 후 연결? (O)

Apply Seq2Seq : Encoder-Decoder



- encoder: 입력된 문장(단어들의 sequence)을 어떤 벡터의 형태(context vector)로 압축 후 hidden state로 decoder에 전달
- decoder: encoder에서 입력받은 벡터와 문장의 시작을 알리는 [start] 를 가지고 다음에 등장할 단어 예상 -> 이후 예상한 단어를 다음 셀의 입력값으로 받아 이전 셀에서의 hidden state와 입력값을 이용해 다음 단어 예상 --> 위와 같은 과정을 문장의 끝을 의미하는 [end]가 다음 단어로 예측될 때까지 반복

encoder-decoder 코드 예시 (번역기)

```
4  import random
5  import torch
6  import torch.nn as nn
7  import torch.optim as optim
```

```
184  SOURCE_MAX_LENGTH = 10
185  TARGET_MAX_LENGTH = 12
186  load_pairs, load_source_vocab, load_target_vocab = preprocess(raw, SOURCE_MAX_LENGTH, TARGET_MAX_LENGTH)
187  print(random.choice(load_pairs))
188
189  enc_hidden_size = 16
190  dec_hidden_size = enc_hidden_size
191  enc = Encoder(load_source_vocab.n_vocab, enc_hidden_size).to(device)
192  dec = Decoder(dec_hidden_size, load_target_vocab.n_vocab).to(device)
193
194  train(load_pairs, load_source_vocab, load_target_vocab, enc, dec, 5000, print_every=1000)
195  evaluate(load_pairs, load_source_vocab, load_target_vocab, enc, dec, TARGET_MAX_LENGTH)
```

=> 184번~195번이 전체 진행 내용을 압축한 코드. 그 이전은 보조함수 구현 역할

#184 => source text(번역해야 할 것)과 target(text)을 train / test set으로 나눈 후 무슨 단어를, 몇 개를 가지고 학습시킬지 정해주는 함수 (문장 최대길이를 제한)

In [2]:

```
import random
import torch
import torch.nn as nn
from torch import optim

torch.manual_seed(0)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

raw = ["I feel hungry.  나는 배가 고프다.",
       "Pytorch is very easy.  파이토치는 매우 쉽다.",
       "Pytorch is a framework for deep learning.  파이토치는 딥러닝을 위한 프레임워크이다.",
       "Pytorch is very clear to use.  파이토치는 사용하기 매우 직관적이다."]

# fix token for "start of sentence" and "end of sentence"
SOS_token = 0
EOS_token = 1
```

- SOS: Start of Sentence. decoder 첫번째 step의 input
- EOS: End of Sentence: 문장이 정해놓은 최대 길이 보다 짧은 경우 문장이 끝났다는 것을 알려주기 위해 삽입

In [3]:

```
# class for vocabulary related information of data
class Vocab:
    def __init__(self):
        self.vocab2index = {"<SOS>": SOS_token, "<EOS>": EOS_token}
        self.index2vocab = {SOS_token: "<SOS>", EOS_token: "<EOS>"}
        self.vocab_count = {}
        self.n_vocab = len(self.vocab2index)

    def add_vocab(self, sentence):
        for word in sentence.split(" "):
            if word not in self.vocab2index:
                self.vocab2index[word] = self.n_vocab
                self.vocab_count[word] = 1
                self.index2vocab[self.n_vocab] = word
                self.n_vocab += 1
            else:
                self.vocab_count[word] += 1
```

In [4]:

```
# filter out the long sentence from source(원문) and target data(번역 이후)
def filter_pair(pair, source_max_length, target_max_length):
    return len(pair[0].split(" ")) < source_max_length and len(pair[1].split(" ")) < target_max_le

# read and preprocess the corpus data
def preprocess(corpus, source_max_length, target_max_length):
    print("reading corpus...")
    pairs = []
    for line in corpus:
        pairs.append([s for s in line.strip().lower().split("Wt")])
    print("Read {} sentence pairs".format(len(pairs)))

    pairs = [pair for pair in pairs if filter_pair(pair, source_max_length, target_max_length)]
    print("Trimmed to {} sentence pairs".format(len(pairs)))

    source_vocab = Vocab()
    target_vocab = Vocab()

    print("Counting words...")
    for pair in pairs:
        source_vocab.add_vocab(pair[0])
        target_vocab.add_vocab(pair[1])
    print("source vocab size =", source_vocab.n_vocab)
    print("target vocab size =", target_vocab.n_vocab)

    return pairs, source_vocab, target_vocab
```

- Vocab(): 단어의 개수 / dictionary 만들어서 넣는다.

  => 학습에 직접적인 영향 X. 모델이 잘 학습할 수 있도록 데이터를 준비해주는 역할

encoder 정의

In [5]:

```
# declare simple encoder
class Encoder(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(Encoder, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)

    def forward(self, x, hidden):
        x = self.embedding(x).view(1, 1, -1)
        x, hidden = self.gru(x, hidden)
        return x, hidden
```

- embedding: input의 size를 hidden size만큼의 벡터로 줄이는 역할
- GRU: 들어오는 / 나가는 dimendion 선언

decoder 정의

In [6]:

```
# declare simple decoder
class Decoder(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(Decoder, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x, hidden):
        x = self.embedding(x).view(1, 1, -1)
        x, hidden = self.gru(x, hidden)
        x = self.softmax(self.out(x[0]))
        return x, hidden
```

- out: embedding 통해 hidden size로 줄어든 상태의 벡터를 입력받고 target text의 차원(output size)으로 변환 시켜주는 layer

학습시키기

In [7]:

```
# 문장을 텐서화
def tensorize(vocab, sentence):
    indexes = [vocab.vocab2index[word] for word in sentence.split(" ")]
    indexes.append(vocab.vocab2index["<EOS>"])
    return torch.Tensor(indexes).long().to(device).view(-1, 1)

# training seq2seq
def train(pairs, source_vocab, target_vocab, encoder, decoder, n_iter, print_every=1000, learning_ra
    loss_total = 0

    encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate)
    decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate)

    training_batch = [random.choice(pairs) for _ in range(n_iter)]
    training_source = [tensorize(source_vocab, pair[0]) for pair in training_batch]
    training_target = [tensorize(target_vocab, pair[1]) for pair in training_batch]

    criterion = nn.NLLLoss()

    for i in range(1, n_iter + 1):
        source_tensor = training_source[i - 1]
        target_tensor = training_target[i - 1]

        encoder_hidden = torch.zeros([1, 1, encoder.hidden_size]).to(device)

        encoder_optimizer.zero_grad()
        decoder_optimizer.zero_grad()

        source_length = source_tensor.size(0)
        target_length = target_tensor.size(0)

        loss = 0

        for enc_input in range(source_length):
            _, encoder_hidden = encoder(source_tensor[enc_input], encoder_hidden)

        decoder_input = torch.Tensor([[SOS_token]]).long().to(device)
        decoder_hidden = encoder_hidden # connect encoder output to decoder input

        for di in range(target_length):
            decoder_output, decoder_hidden = decoder(decoder_input, decoder_hidden)
            loss += criterion(decoder_output, target_tensor[di])
            decoder_input = target_tensor[di]  # teacher forcing

        loss.backward()

        encoder_optimizer.step()
        decoder_optimizer.step()

        loss_iter = loss.item() / target_length
        loss_total += loss_iter

        if i % print_every == 0:
            loss_avg = loss_total / print_every
            loss_total = 0
            print("[{} - {}%] loss = {:05.4f}".format(i, i / n_iter * 100, loss_avg))
```

- tensorize: 문장을 one-hot encoding 통해 텐서화 시키는 함수

- NLLLoss: category value끼리 비교할 때 많이 사용되는 loss function (CrossEntrophy도 사용가능)
- teacher forcing: 이전 셀의 예측값을 다음 셀에 넣는 대신 직접 정답을 넣어 더 빨리 수렴하게 하는 방법 (학습과정에서만 쓰이고 test에서는 X)

  => 학습이 불안정해진다는 부작용 (random하게 일정%만 할 수도)

모델 평가

In [8]:

```
# insert given sentence to check the training
def evaluate(pairs, source_vocab, target_vocab, encoder, decoder, target_max_length):
    for pair in pairs:
        print(">", pair[0])
        print("=", pair[1])
        source_tensor = tensorize(source_vocab, pair[0])
        source_length = source_tensor.size()[0]
        encoder_hidden = torch.zeros([1, 1, encoder.hidden_size]).to(device)

        for ei in range(source_length):
            _, encoder_hidden = encoder(source_tensor[ei], encoder_hidden)

        decoder_input = torch.Tensor([[SOS_token]], device=device).long()
        decoder_hidden = encoder_hidden
        decoded_words = []

        for di in range(target_max_length):
            decoder_output, decoder_hidden = decoder(decoder_input, decoder_hidden)
            _, top_index = decoder_output.data.topk(1)
            if top_index.item() == EOS_token:
                decoded_words.append("<EOS>")
                break
            else:
                decoded_words.append(target_vocab.index2vocab[top_index.item()])

            decoder_input = top_index.squeeze().detach()

        predict_words = decoded_words
        predict_sentence = " ".join(predict_words)
        print("<", predict_sentence)
        print("")
```

In [9]:

```
SOURCE_MAX_LENGTH = 10
TARGET_MAX_LENGTH = 12

load_pairs, load_source_vocab, load_target_vocab = preprocess(raw, SOURCE_MAX_LENGTH, TARGET_MAX_LEN
print(random.choice(load_pairs))
```

```
reading corpus...
Read 4 sentence pairs
Trimmed to 4 sentence pairs
Counting words...
source vocab size = 17
target vocab size = 13
['pytorch is a framework for deep learning.', '파이토치는 딥러닝을 위한 프레임워크이
다.']
```

In [10]:

```
# declare the encoder and the decoder
enc_hidden_size = 16
dec_hidden_size = enc_hidden_size
enc = Encoder(load_source_vocab.n_vocab, enc_hidden_size).to(device)
dec = Decoder(dec_hidden_size, load_target_vocab.n_vocab).to(device)
```

In [11]:

```
# train seq2seq model
train(load_pairs, load_source_vocab, load_target_vocab, enc, dec, 5000, print_every=1000)
```

```
[1000 - 20.0%] loss = 0.7413
[2000 - 40.0%] loss = 0.1081
[3000 - 60.0%] loss = 0.0334
[4000 - 80.0%] loss = 0.0182
[5000 - 100.0%] loss = 0.0125
```

In [12]:

```
# check the model with given data
evaluate(load_pairs, load_source_vocab, load_target_vocab, enc, dec, TARGET_MAX_LENGTH)
```

```
> i feel hungry.
= 나는 배가 고프다.
< 나는 배가 고프다. <EOS>

> pytorch is very easy.
= 파이토치는 매우 쉽다.
< 파이토치는 매우 쉽다. <EOS>

> pytorch is a framework for deep learning.
= 파이토치는 딥러닝을 위한 프레임워크이다.
< 파이토치는 딥러닝을 위한 프레임워크이다. <EOS>

> pytorch is very clear to use.
= 파이토치는 사용하기 매우 직관적이다.
< 파이토치는 사용하기 매우 직관적이다. <EOS>
```

한계

1. 정보손실: 하나의 고정된 크기의 벡터에 모든 정보를 압축하는 과정에서 정보 손실 불가피
2. Vanishing Gradient: RNN의 고질적인 문제 발생

한가지 대안: Attention Mechanism

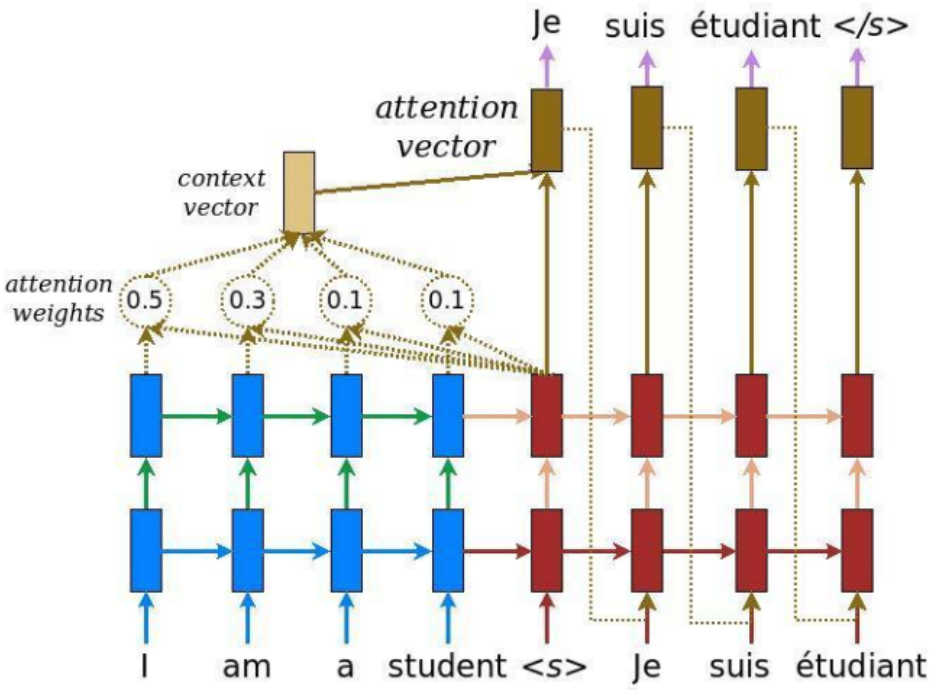
In [14]:

```
from IPython.display import Image
```

In [15]:

```
Image("seq6.png",width=700, height=700)
```

Out[15]:



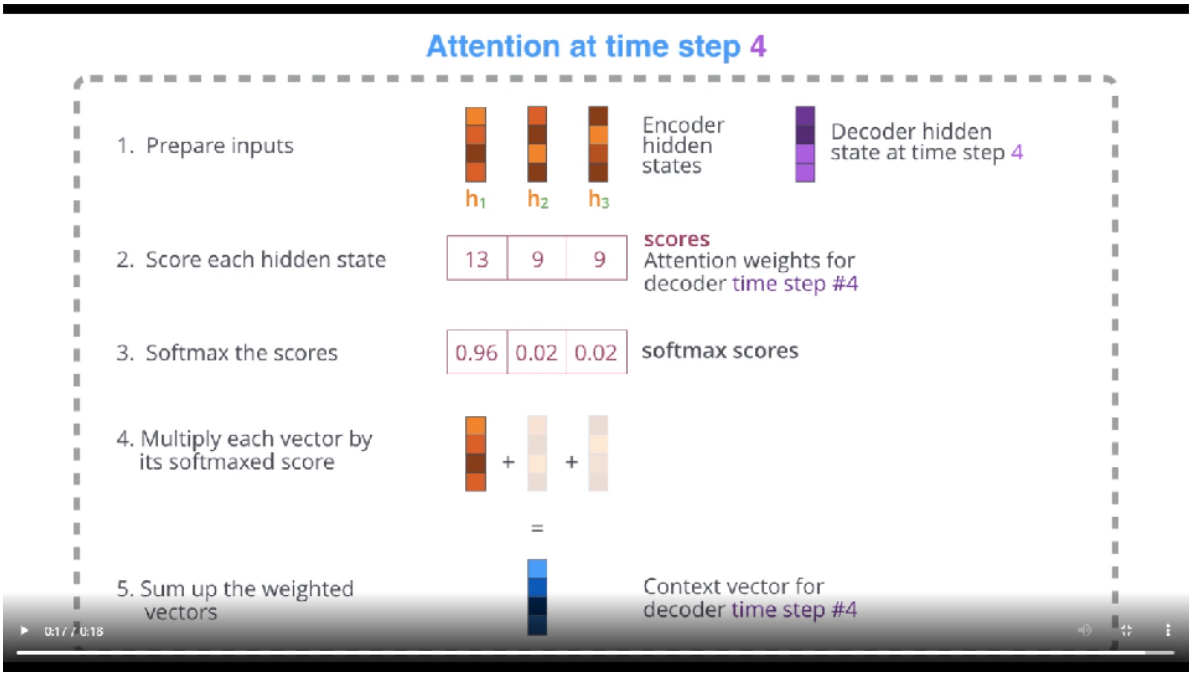
①. 목적: Decoder에서 출력 단어를 예측하는 매 시점(time step)마다, Encoder에서의 전체 입력 문장을 다시 한 번 참고

(단, 전체 입력 문장을 전부 다 동일한 비율로 참고하는 것이 아니라, 해당 시점에서 예측해야 할 단어와 연관이 있는 입력 단어 부분을 좀 더 집중(attention)해서 봄)

In [16]:

```
Image('attention_process.png', width=800, height=800)
```

Out[16]:



②. Process

1. attention score를 구한다.

(encoder의 모든 hidden state에 대해, decoder의 t-1 시점의 hidden state 와의 유사도 계산)

In [17]:

```
Image('seq7.png')
```

Out[17]:

이름	스코어 함수	Defined by
dot	$score(s_t,\ h_i) = s_t^T h_i$	Luong et al. (2015)
$scaled\ dot$	$score(s_t,\ h_i) = \frac{s_t^T h_i}{\sqrt{n}}$	Vaswani et al. (2017)
$general$	$score(s_t,\ h_i) = s_t^T W_a h_i$ // 단, W_a는 학습 가능한 가중치 행렬	Luong et al. (2015)
$concat$	$score(s_t,\ h_i) = W_a^T\ tanh(W_b[s_t; h_i])$	Bahdanau et al. (2015)
$location - base$	$\alpha_t = softmax(W_a s_t)$ // α_t 산출 시에 s_t만 사용하는 방법.	Luong et al. (2015)

위에서 s_t는 Query, h_i는 Keys, W_a와 W_b는 학습 가능한 가중치 행렬입니다.

2. softmax 함수를 사용하여 attention distribution을 구한다. (각각의 softmax 값 출력)

3. attention distribution를 이용하여, encoder의 모든 hidden states를 가중합한다. (이를 context vector 라고 함)

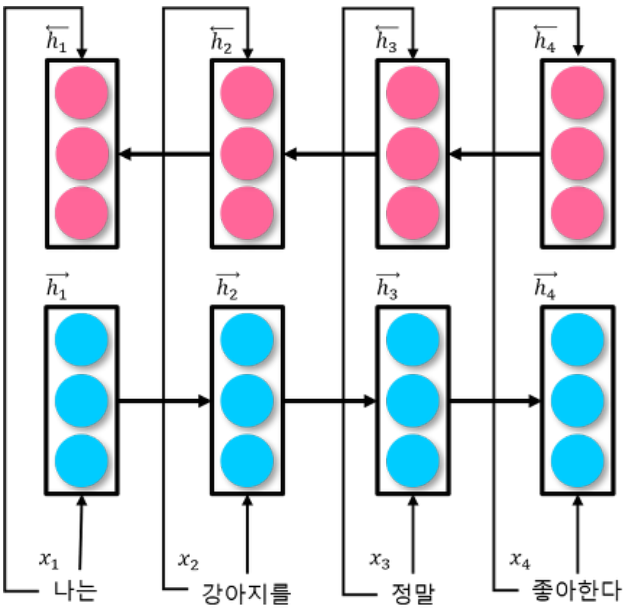
4. t 시점의 decoder 출력한다. [f(t-1 시점의 hidden state, t-1 시점의 decoder 출력, t 시점의 context vector)]

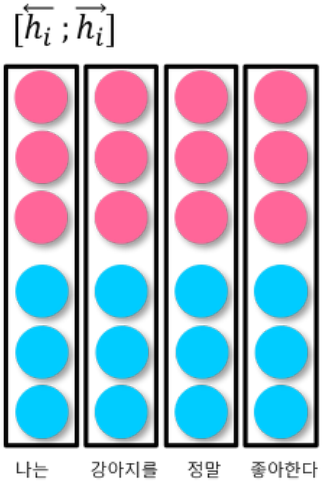
③. Example

In [18]:

```
Image('encoder.png',width=700, height=700) #encoder: hidden states 생성
```

Out[18]:

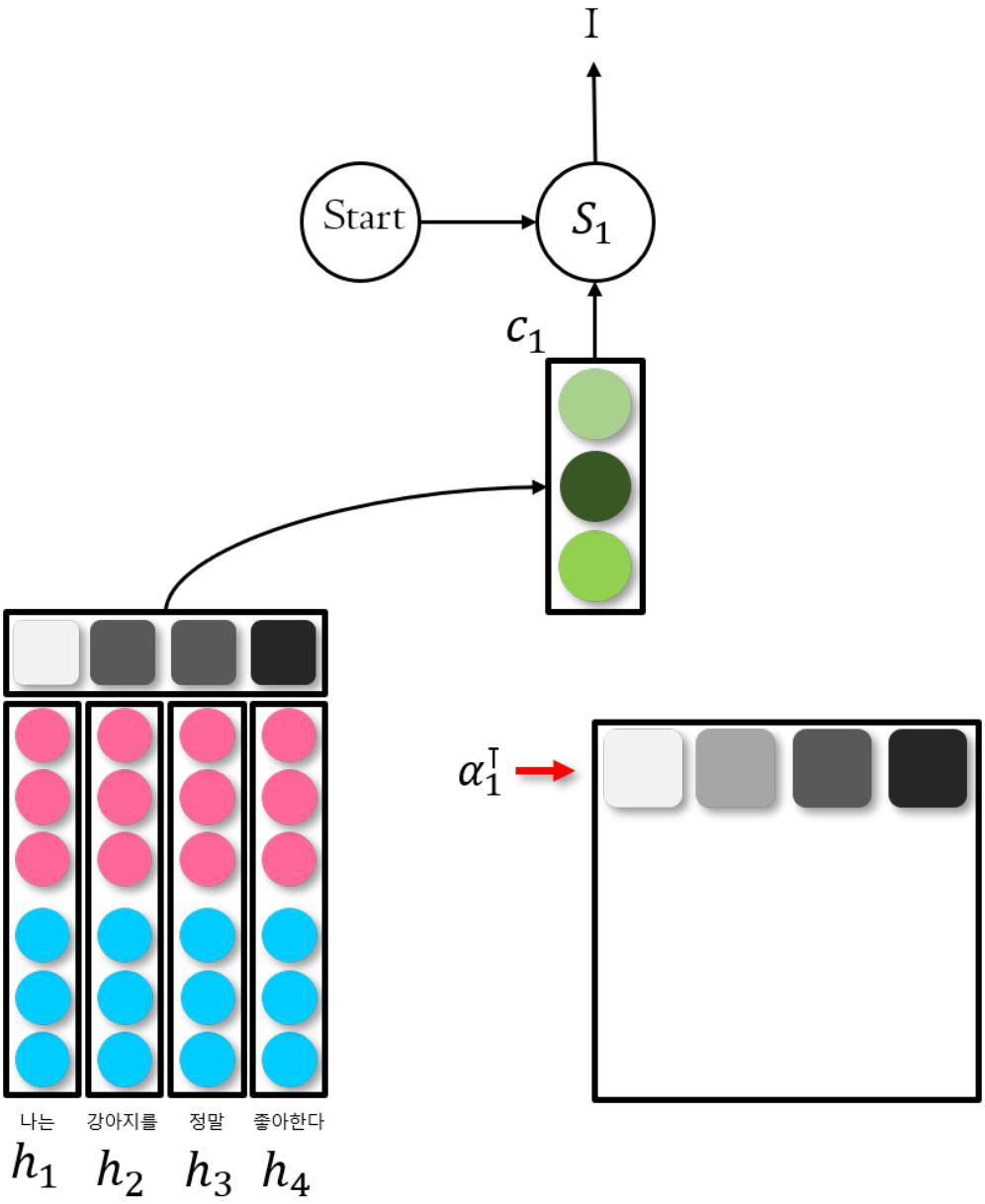




In [21]:

```
Image('deco1.png',width=500, height=50)
```

Out[21]:



1. 유사도 계산(alpha) : 흰색일수록 유사도 높음 (softmax 함수까지 적용한 결과)

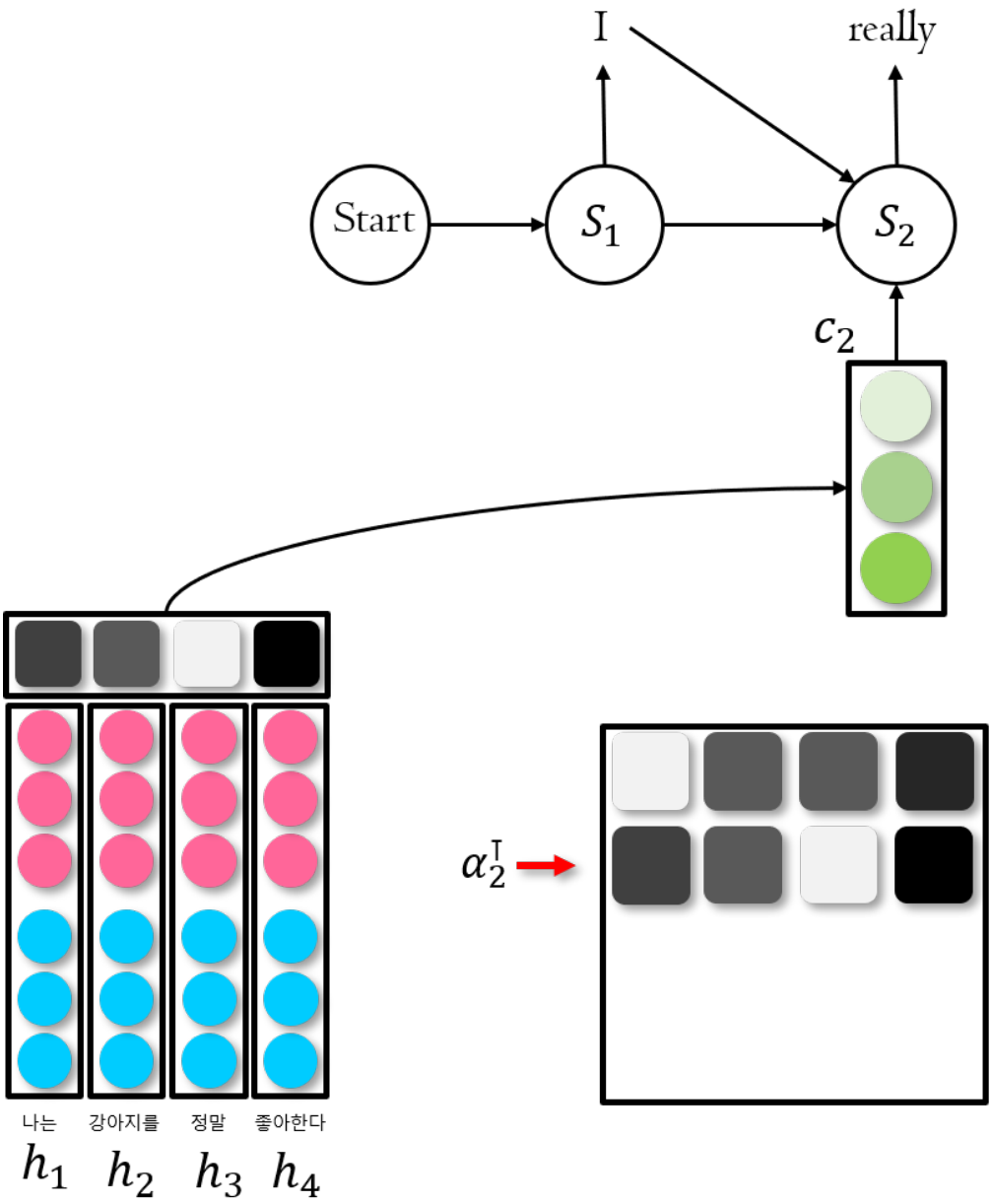
2. encoder의 hidden states와 alpha를 dot 하여 가중합 -> context vector 완성

3. 1st decoder 출력값을 구하는 것이므로, 입력값은 2개 (Encoder에서의 last hidden state와 context vector)

In [20]:

```
Image('deco2.png',width=500, height=500)
#function(이전 hidden state, 이전 decoder 출력, context vector)
```

Out[20]:



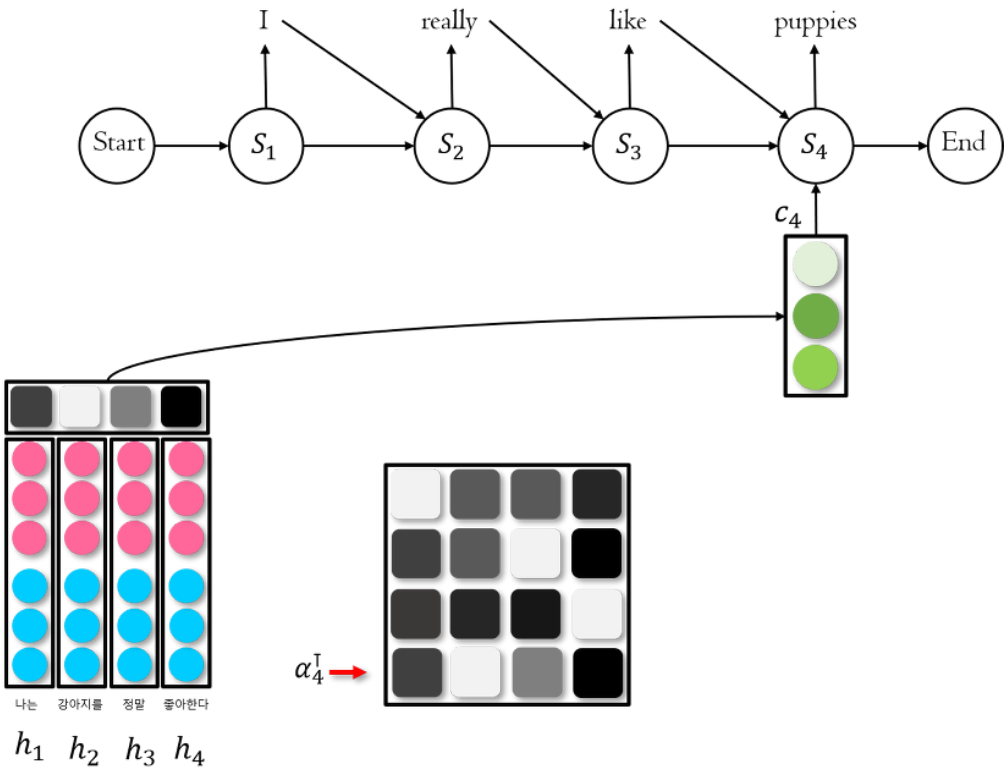
4. 같은 방식으로 반복!

(여기서부터는 t-1시점의 decoder 출력값까지 포함하여, 총 3개의 입력값으로 t 시점의 decoder 출력값 예측)

In [22]:

```
Image('deco4.png', width=500, height=500) #반복
```

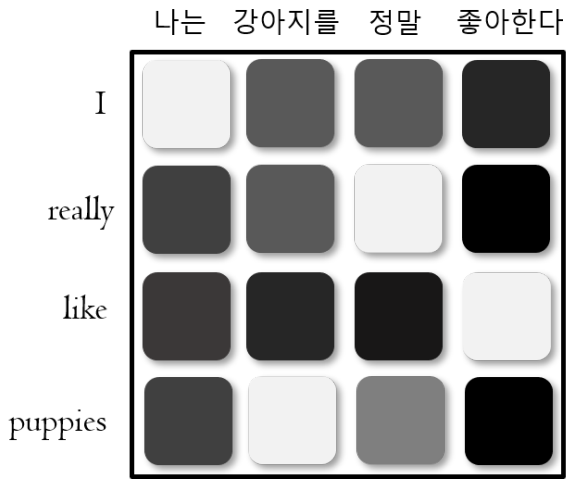
Out[22]:



In [23]:

```
Image("deco3.png",width=300, height=300) #완성!
```

Out[23]:



- 출처

https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/ (https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/)

https://glee1228.tistory.com/3 (https://glee1228.tistory.com/3)

https://wikidocs.net/22893 (https://wikidocs.net/22893)

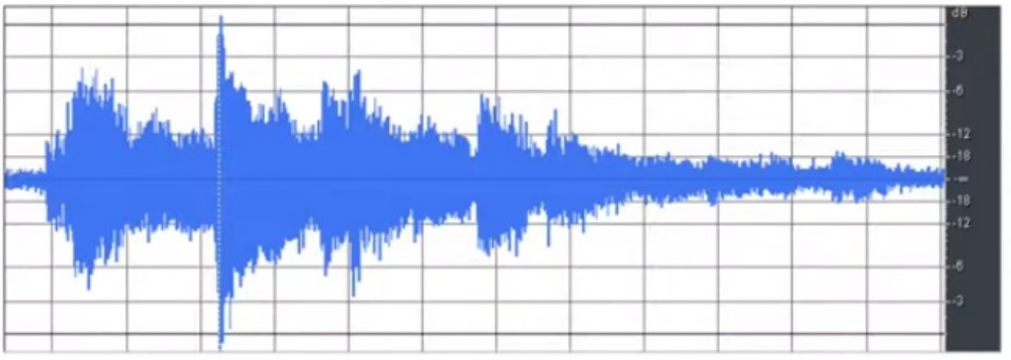
https://nlpstudynote.tistory.com/18 (https://nlpstudynote.tistory.com/18)

Lab 11- 6 PackedSequence

각각 길이가 다른 **Sequence** 데이터를 하나의 **batch**에 넣는 방식

Sequential data

- Text
  - "The quick brown fox jumps over the lazy dog"
- Audio



https://commons.wikimedia.org/wiki/File:ALC_orig.png

계속 **Sequential data**의 예시로 **text** 데이터를 많이 다루어 왔다. 그리고 오디오 데이터도 **sequential data**이다. 그러나 생각해 보자면 위 문장, **audio**처럼 모든 **data**가 같은 길이일리가..

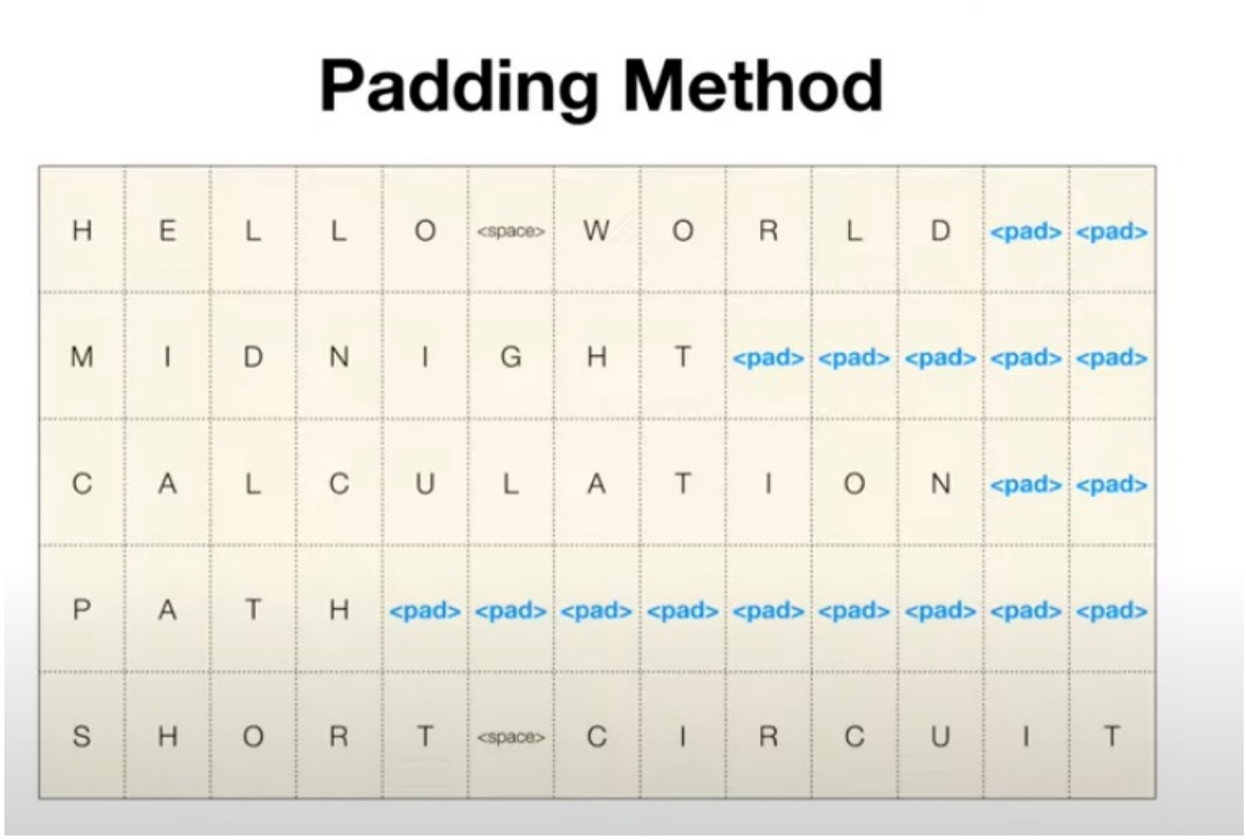
이 **length difference** 문제는 이미지 (가령 **3, 28, 28, fixed size**)를 다룰 때와 상반된다

지금 문제로 돌아오자

Example data

H	E	L	L	O	<space>	W	O	R	L	D	
M	I	D	N	I	G	H	T				
C	A	L	C	U	L	A	T	I	O	N	
P	A	T	H								
S	H	O	R	T	<space>	C	I	R	C	U	I	T

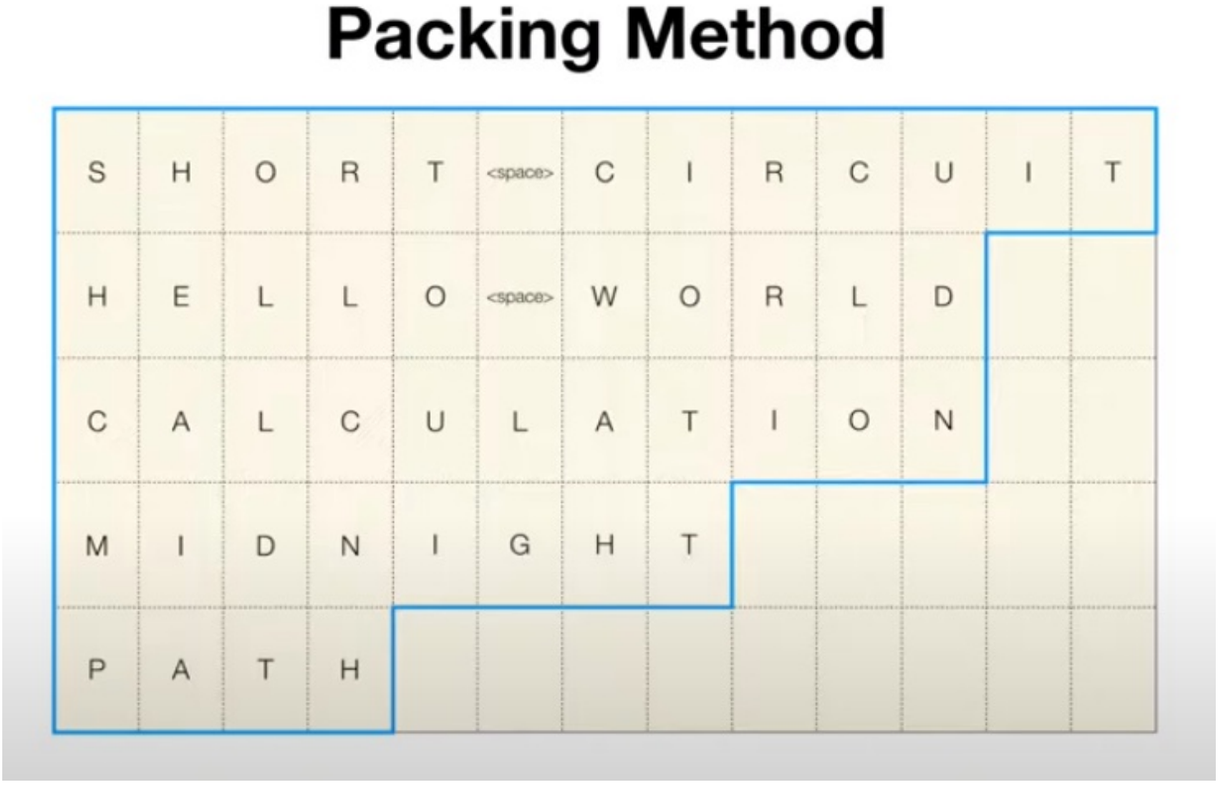
Sample size = 5인 sequential data를 예로 든다. 길이가 다른 문제를 해결하는 **packedsequence**에는 두 방식이 있다



1. Padding : "덧대다"

https://en.dict.naver.com/#/entry/enko/9dfd90135bf248389295e6345f7f82e8

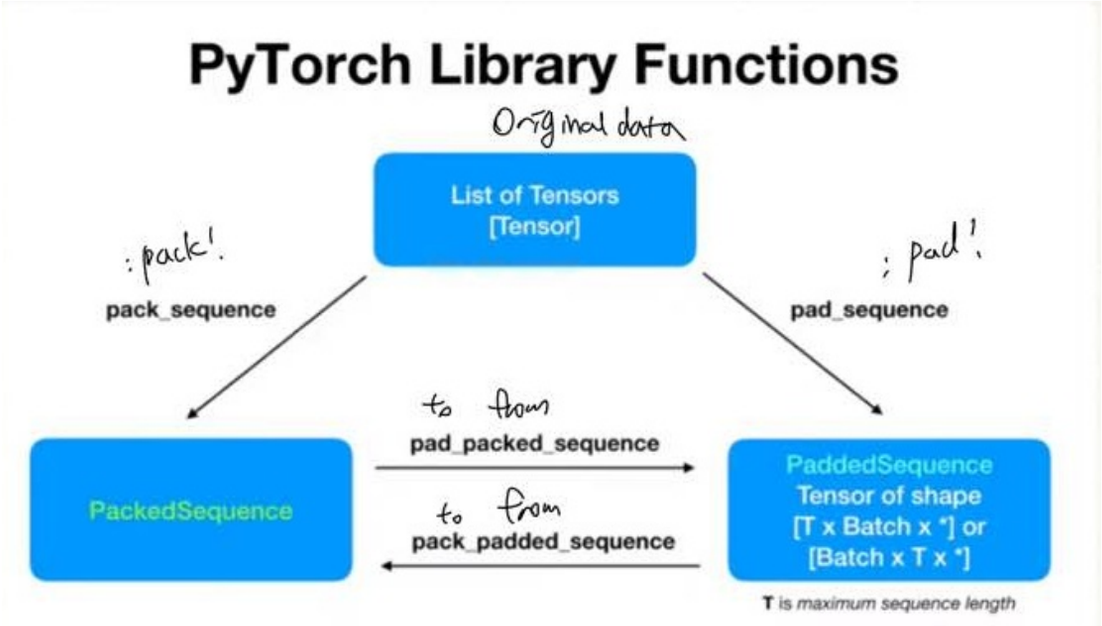
- 가장 긴 sequence data의 길이를 기준으로 pad 토큰으로 채운다
- token : 의미를 가지는 최소한의 문자 덩어리(문자열) http://www.ktword.co.kr/abbr_view.php?m_temp1=3673
- Plus : tensor 변환 가능
- Minus : 계산량 불필요하게 증가



2. Packing : "묶다", "포장하다"

https://en.dict.naver.com/#/search?query=pack

- Pytorch 구현 위해서는 길이가 긴 순서로.
- Plus : padding 불필요
- minus : 내림차순 (길이) 정렬 연산.



실습코드를 알아보자고 하였으나 종강해서 코드를 집어넣었다

Lab 11-6.5 : Code

overall source : https://simonjisu.github.io/nlp/2018/07/05/packedsequence.html#pytorch---packedsequence

1st) Pad the data!

In [1]:

```
import torch
import torch.nn as nn
import numpy as np
np.random.seed(2020)
```

In [2]:

```
batch_data = ["I thought by eliminating half of life the other half would thrive but you've shown
me that's impossible",
              "Hunlearning is a youtube channel having two million subscribers",
              "I hope to travel abroad one day",
              "The more I suffer, the more I love her",
              "You either die a hero or you live long enough to see yourself become the villain"]
```

In [3]:

```
input_seq = [s.split() for s in batch_data]; input_seq
```

Out[3]:

```
[['I',
  'thought',
  'by',
  'eliminating',
  'half',
  'of',
```

```
  'life',
  'the',
  'other',
  'half',
  'would',
  'thrive',
  'but',
  "you've",
  'shown',
  'me',
  "that's",
  'impossible'],
 ['Hunlearning',
  'is',
  'a',
  'youtube',
  'channel',
  'having',
  'two',
  'million',
  'subscribers'],
 ['I', 'hope', 'to', 'travel', 'abroad', 'one', 'day'],
 ['The', 'more', 'I', 'suffer,', 'the', 'more', 'I', 'love', 'her'],
 ['You',
  'either',
  'die',
  'a',
  'hero',
  'or',
  'you',
  'live',
  'long',
  'enough',
  'to',
  'see',
  'yourself',
  'become',
  'the',
  'villain']]
```

In [4]:

```
max_len = 0
for s in input_seq:
    if len(s) >= max_len:
        max_len = len(s)

print(max_len) # 이것을 기준으로 내림차순
print(batch_data[np.argmax(max_len)])
```

```
18
I thought by eliminating half of life the other half would thrive but you've shown me that's
impossible
```

In [5]:

```
vocab = {w: i for i, w in enumerate(set([t for s in input_seq for t in s]), 1)}
vocab["<pad>"] = 0
print(vocab) # 빈도가 아니라 index임에 유의!
```

```
{'of': 1, 'thrive': 2, 'enough': 3, 'thought': 4, 'day': 5, 'become': 6, 'Hunlearning': 7,
'travel': 8, 'a': 9, 'million': 10, 'to': 11, 'You': 12, "you've": 13, 'her': 14, 'channel': 15, '
you': 16, 'is': 17, 'me': 18, 'would': 19, 'impossible': 20, 'having': 21, 'half': 22, 'or': 23, '
villain': 24, 'youtube': 25, 'love': 26, 'hero': 27, 'the': 28, 'shown': 29, "that's": 30, 'other'
: 31, 'yourself': 32, 'eliminating': 33, 'two': 34, 'more': 35, 'die': 36, 'hope': 37, 'I': 38, 's
ubscribers': 39, 'one': 40, 'long': 41, 'see': 42, 'abroad': 43, 'The': 44, 'by': 45, 'either': 46
, 'but': 47, 'life': 48, 'suffer,': 49, 'live': 50, '<pad>': 0}
```

In [6]:

```
input_seq = [s+["<pad>"]*(max_len-len(s)) if len(s) < max_len else s for s in input_seq]

input_seq2idx = torch.LongTensor([list(map(vocab.get, s)) for s in input_seq])
```

```
input_seq # 가장 긴 것을 기준으로 나머지는 padded
```

Out[6]:

```
[['I',
  'thought',
  'by',
  'eliminating',
  'half',
  'of',
  'life',
  'the',
  'other',
  'half',
  'would',
  'thrive',
  'but',
  "you've",
  'shown',
  'me',
  "that's",
  'impossible'],
 ['Hunlearning',
  'is',
  'a',
  'youtube',
  'channel',
  'having',
  'two',
  'million',
  'subscribers',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>'],
 ['I',
  'hope',
  'to',
  'travel',
  'abroad',
  'one',
  'day',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>'],
 ['The',
  'more',
  'I',
  'suffer,',
  'the',
  'more',
  'I',
  'love',
  'her',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>',
  '<pad>'],
```

```
 ['You',
  'either',
  'die',
  'a',
  'hero',
  'or',
  'you',
  'live',
  'long',
  'enough',
  'to',
  'see',
  'yourself',
  'become',
  'the',
  'villain',
  '<pad>',
  '<pad>']]
```

In [7]:

```
input_seq2idx # encoded, pad 자리에 index 0이 들어감
```

Out[7]:

```
tensor([[38,  4, 45, 33, 22,  1, 48, 28, 31, 22, 19,  2, 47, 13, 29, 18, 30, 20],
        [ 7, 17,  9, 25, 15, 21, 34, 10, 39,  0,  0,  0,  0,  0,  0,  0,  0,  0],
        [38, 37, 11,  8, 43, 40,  5,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
        [44, 35, 38, 49, 28, 35, 38, 26, 14,  0,  0,  0,  0,  0,  0,  0,  0,  0],
        [12, 46, 36,  9, 27, 23, 16, 50, 41,  3, 11, 42, 32,  6, 28, 24,  0,  0]])
```

2nd) Pack the Padded Sequence using "pack_padded_sequence"

In [8]:

```
from torch.nn.utils.rnn import pack_padded_sequence
```

In [9]:

```
input_lengths = torch.LongTensor([torch.max(input_seq2idx[i, :].data.nonzero())+1
                                  for i in range(input_seq2idx.size(0))])
input_lengths, sorted_idx = input_lengths.sort(0, descending=True) # 내림차순
input_seq2idx = input_seq2idx[sorted_idx]
```

In [10]:

```
input_lengths
```

Out[10]:

```
tensor([18, 16,  9,  9,  7])
```

In [11]:

```
input_seq2idx
```

Out[11]:

```
tensor([[38,  4, 45, 33, 22,  1, 48, 28, 31, 22, 19,  2, 47, 13, 29, 18, 30, 20],
        [12, 46, 36,  9, 27, 23, 16, 50, 41,  3, 11, 42, 32,  6, 28, 24,  0,  0],
        [ 7, 17,  9, 25, 15, 21, 34, 10, 39,  0,  0,  0,  0,  0,  0,  0,  0,  0],
        [44, 35, 38, 49, 28, 35, 38, 26, 14,  0,  0,  0,  0,  0,  0,  0,  0,  0],
        [38, 37, 11,  8, 43, 40,  5,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]])
```

In [12]:

```
packed_input = pack_padded_sequence(input_seq2idx, input_lengths.tolist(), batch_first=True)
```

In [13]:

```
print(type(packed_input))
print(packed_input[0])  # packed data
print(packed_input[1])  # batch_sizes
```

```
<class 'torch.nn.utils.rnn.PackedSequence'>
tensor([38, 12,  7, 44, 38,  4, 46, 17, 35, 37, 45, 36,  9, 38, 11, 33,  9, 25,
        49,  8, 22, 27, 15, 28, 43,  1, 23, 21, 35, 40, 48, 16, 34, 38,  5, 28,
        50, 10, 26, 31, 41, 39, 14, 22,  3, 19, 11,  2, 42, 47, 32, 13,  6, 29,
        28, 18, 24, 30, 20])
tensor([5, 5, 5, 5, 5, 5, 5, 4, 4, 2, 2, 2, 2, 2, 2, 2, 1, 1])
```

3rd) RNN 모델 훈련 (LSTM, GRU에서도 당연 쓸수있다)

여기부터는 아까 배운 바와 동일하다

In [14]:

```
vocab_size = len(vocab)
hidden_size = 1
embedding_size = 5
num_layers = 3
```

In [15]:

```
embed = nn.Embedding(vocab_size, embedding_size, padding_idx=0)
gru = nn.RNN(input_size=embedding_size, hidden_size=hidden_size, num_layers=num_layers,
             bidirectional=False, batch_first=True)
```

In [16]:

```
embeded = embed(input_seq2idx)
packed_input = pack_padded_sequence(embeded, input_lengths.tolist(), batch_first=True)
packed_output, hidden = gru(packed_input)
```

In [17]:

```
packed_output[0].size(), packed_output[1]
```

Out[17]:

```
(torch.Size([59, 1]),
 tensor([5, 5, 5, 5, 5, 5, 5, 4, 4, 2, 2, 2, 2, 2, 2, 2, 1, 1]))
```

In [18]:

```
packed_output
```

Out[18]:

```
PackedSequence(data=tensor([[-0.0771],
        [-0.2568],
        [-0.1637],
        [ 0.0055],
        [-0.0771],
        [-0.2012],
        [-0.2010],
        [-0.1614],
        [-0.1263],
        [-0.1931],
        [-0.1439],
        [-0.3795],
        [-0.2694],
        [-0.1795],
        [-0.2272],
        [-0.1149],
        [-0.3101]
```

```
        [-0.3101],
        [-0.2746],
        [-0.2096],
        [-0.4256],
        [-0.1165],
        [-0.4706],
        [-0.2291],
        [-0.2504],
        [-0.3244],
        [-0.3470],
        [-0.4775],
        [-0.3816],
        [-0.2736],
        [-0.4670],
        [-0.2834],
        [-0.4711],
        [-0.3077],
        [-0.3560],
        [-0.3573],
        [-0.3478],
        [-0.4881],
        [-0.4317],
        [-0.3090],
        [-0.3445],
        [-0.5368],
        [-0.5065],
        [-0.4469],
        [-0.2875],
        [-0.4342],
        [-0.4554],
        [-0.5028],
        [-0.3992],
        [-0.5272],
        [-0.3343],
        [-0.6072],
        [-0.4465],
        [-0.5633],
        [-0.4129],
        [-0.4820],
        [-0.3727],
        [-0.5700],
        [-0.4276],
        [-0.3721]], grad_fn=<CatBackward>), batch_sizes=tensor([5, 5, 5, 5, 5, 5, 5, 4, 4, 2, 2, 2,
2, 2, 2, 2, 1, 1]), sorted_indices=None, unsorted_indices=None)
```

4th) 모델 output을 얻기 위해 unpacking

In [19]:

```
from torch.nn.utils.rnn import pad_packed_sequence
```

In [20]:

```
output, output_lengths = pad_packed_sequence(packed_output, batch_first=True)
output.size(), output_lengths
```

Out[20]:

```
(torch.Size([5, 18, 1]), tensor([18, 16,  9,  9,  7]))
```

In [21]:

```
len(packed_output[0])
```

Out[21]:

```
59
```

위에 것들은 unique, 아래것들은 padded. (tensor를 모양 fitting 위해)

In [22]:

```
output.shape
```

Out[22]:

```
torch.Size([5, 18, 1])
```

In [23]:

```
import pandas as pd
```

In [24]:

```
def color_white(val):
    color = 'blue' if val == 0 else 'black'
    return 'color: {}'.format(color)
def color_red(data):
    max_len = len(data)
    fmt = 'color: red'
    lst = []
    for i, v in enumerate(data):
        if (v != 0) and (i == max_len-1):
            lst.append(fmt)
        elif (v != 0) and (data[i+1] == 0):
            lst.append(fmt)
        else:
            lst.append('')
    return lst
```

In [25]:

```
df = pd.DataFrame(np.concatenate([o.detach().numpy() for o in output.transpose(0, 1)], axis=1).roun
d(4))
df.index.name = 'batch'
df.columns.name = 'hidden_step'
df.style.applymap(color_white).apply(color_red, axis=1)
```

Out[25]:

hidden_step batch	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	-0.0771	-0.2012	-0.1439	-0.1149	-0.1165	-0.347	-0.2834	-0.3478	-0.3445	-0.2875	-0.4554	-0.3992	-0.3343	-0.4465	-0.4129
1	-0.2568	-0.201	-0.3795	-0.3101	-0.4706	-0.4775	-0.4711	-0.4881	-0.5368	-0.4342	-0.5028	-0.5272	-0.6072	-0.5633	-0.482
2	-0.1637	-0.1614	-0.2694	-0.2746	-0.2291	-0.3816	-0.3077	-0.4317	-0.5065	0	0	0	0	0	0
3	0.0055	-0.1263	-0.1795	-0.2096	-0.2504	-0.2736	-0.356	-0.309	-0.4469	0	0	0	0	0	0
4	-0.0771	-0.1931	-0.2272	-0.4256	-0.3244	-0.467	-0.3573	0	0	0	0	0	0	0	0

빨강 : 각 sequence에서의 마지막 hidden vector

파랑 : pad

https://dl.dropbox.com/s/jl1iymxj6fdtvoe/0705img4.gif