20.06.29

주제 **Time Series Data / Seq2Seq / Sequence Data**

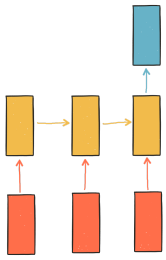
시계열 데이터(Time Series Data)

시계열(time series) 데이터란 시간과 관련이 있는 데이터를 말한다.
즉 각 데이터들이 시간을 기준으로 선후관계를 가진다.

특히 대부분의 경우 시간을 기준으로 의존성을 가지게 된다.
여기서 의존성이란 시간 t에 발생한 데이터는 시간 t-1의 데이터로 부터 의존적이라는 의미.

주가, 환율, 기후변화 데이터 등등.

이를 RNN으로 학습시키는 모델은 many to one 꼴이 된다.



각 학습의 단계마다 이전 시점의 데이터를 반영하여 특정 시점의 나타날 데이터를 예측하는 것이다.

이 때 얻고자 하는 결과가 시계열 데이터의 특정 feature라면 결과가 디멘션 1짜리 벡터로 나오게 되는데 만약 마지막 셀에서 연산한 결과를 그대로 출력으로 이용하려면 다음 셀로 data를 유통하기 위한 hidden state도 하나의 디멘션이 되어야 한다. 그런데 시계열 데이터를 제대로 학습하기 위한 데이터의 feature들은 통상 그 수가 매우 많고 복잡하므로 이는 모델에 큰 부담이 된다. 그러므로 hidden state에 여러개의 디멘션을 보장해주고 마지막 단계에서 FC layer를 연결하여 출력값을 결괏값으로 한는 방법이 일반적이라고 한다.

실습 코드는 크게 특이할 건 없는 듯 하다.
변수 정하고, LSTM, criterion, optimizer 등을 적절히 선택하여 학습을 진행하면 된다.
데이터 feature 간 크기 차이가 많이 나면 이 또한 모델에 부담이 되므로 스케일링 한다.

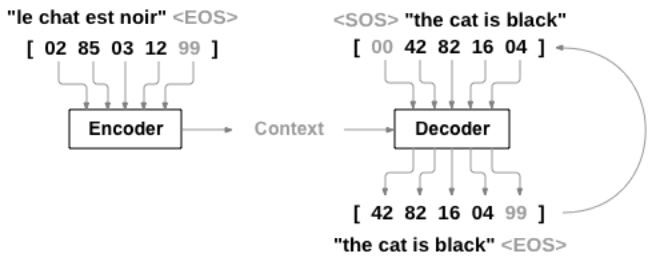
시계열 데이터를 학습하고 예측하는데 있어 모델보다는 어떤 feature를 선택할 것인지가 더 중요해 보인다. 시계열 데이터 해석과 예측을 위한 방법론은 상경계열 대학원생들이 배우는 것 같다.

Seq2Seq

https://tutorials.pytorch.kr/intermediate/seq2seq_translation_tutorial.html

RNN을 사용하다 보면 각각의 입력에 대한 즉각적인 출력을 하곤 한다. 그러다 보니 '문맥' 이라는 것을 제대로 못 읽고, 함축적인 의미를 담

고 있거나 끝에 가서야 의도가 드러나는 문장을 해석하기는 어려워한다. 이를 해결하기 위해 RNN 두개를 붙여 설계된 모델이 Seq2Seq이다.



Seq2Seq 네트워크는 Encoder Decoder network 라고도 한다.
인코더 및 디코더라고 하는 두 개의 RNN으로 구성되어 있기 때문이다.

인코더는 입력 시퀀스를 읽어 이 정보를 하나의 벡터를 생성하여 표현하고,
디코더는 해당 벡터를 첫번째 hidden state로 삼아 출력 시퀀스를 생성한다.

모든 입력에 해당하는 출력이 있는 단일 RNN의 시퀀스 예측과 달리
Seq2Seq 모델은 시퀀스 길이와 순서가 자유로우므로 번역등의 작업에 이상적이다.

언어마다 문법과 어순에 차이가 있으므로 단순히 단어들을 직역하여 치환하는 것만으로는
정확한 번역을 하기 어렵다.

이상적인 경우에 입력 시퀀스의 '문맥'을
문장의 N 차원 공간에 있는 단일 지점인 단일 벡터으로 인코딩할 수 있다.

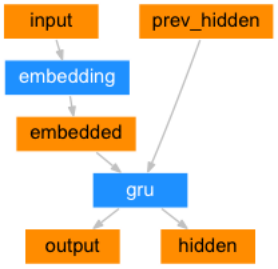
Seq2Seq 모델을 설정하고 학습시키는 전반적인 코드 구조는 다음과 같다.

```
# Preprocess
SOURCE_MAX_LENGTH = 10 # Source 문장의 최대 길이
TARGET_MAX_LENGTH = 12 # Target 문장의 최대 길이
# raw라는 원문을 받아서 source와 Target으로 나누고, 이를 training과 test set으로 또 나눈다.
load_pairs, load_source_vocab, load_target_vocab = preprocess(raw, SOURCE_MAX_LENGTH, TARGET_MAX_LENGTH)

## 중략...

# Encoder와 Decoder라는 두 개의 RNN Layer를 구성한다.
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

# 학습과 평가
train(load_pairs, load_source_vocab, load_target_vocab, enc, dec, 5000, print__every=1000)
evaluate(load_pairs, load_source_vocab, load_target_vocab, enc, dec, TARGET_MAX_LENGTH)
```

인코더

Seq2Seq 네트워크의 인코더는 입력 문장의 모든 단어에 대해 어떤 값을 출력하는 RNN이다.

모든 입력 단어에 대해 인코더는 벡터와 은닉 상태를 출력하고
다음 입력 단어를 위해 그 은닉 상태를 사용한다.



인코더는 코드로 구현한다면 이렇다.

```
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
```

embedding을 통과한 x를 다시 gru로 넣어 return 을 받게 되는 구조다.

gru, Embedding 의 경우엔 그냥 nn 에서 가져오면 된다. Embedding은 거대한 행렬이라고 생각하면 된다. input을 넣을 땐 source text를 이루고 있는 단어의 개수대로 one hot encoding을 해주는데, one-hot-encoding된 vector(NX1)를 EMB 행렬(MXN)과 곱하여 Hidden size만큼의 차원으로(MX1) 줄여주는 기능을 한다 (MXN 행렬곱 NX1 = MX1). 줄어든 Vector는 GRU에 들어와 처리가 된다.

디코더

디코더는 인코더 출력 벡터를 받아서 번역을 생성하기 위한 단어 시퀀스를 출력한다.

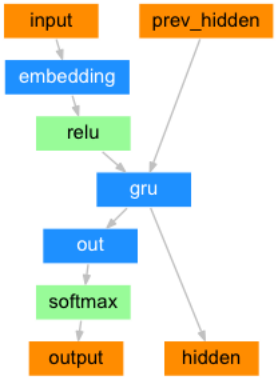
```
class Decoder(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(Decoder, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, 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
```

위 코드에 대한 간단한 설명을 하자면, GRU에 들어가는 것은 one-hot-vector를 Embedding 처리해 원하는 사이즈(Hidden size)로 줄여준 벡터다. Hidden size 만한 벡터를 GRU에서 처리해 hidden size만한 output이 나오는데, Target Text에 사용되는 단어로 복원하기 위해서 out이라는 layer(nn.Linear)를 거쳐 다시 vector 크기를 out에 맞춰서 늘려 준다.

간단한 디코더

가장 간단한 Seq2Seq 디코더는 인코더의 마지막 출력만을 이용한다.
이 마지막 출력은 전체 시퀀스에서 문맥을 인코드하기 때문에 문맥 벡터(context vector) 로 불린다. 이 문맥 벡터는 디코더의 초기 은닉 상태로 사용 된다.

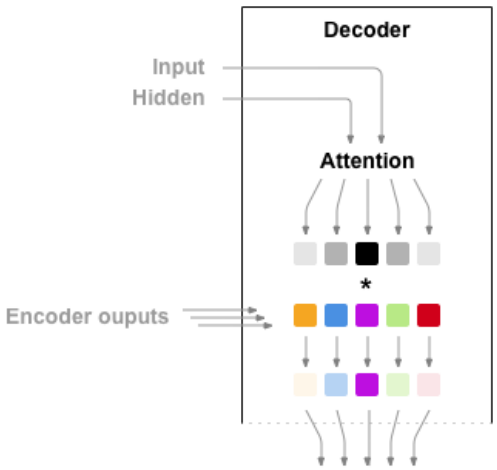
디코딩의 매 단계에서 디코더에게 입력 토큰과 은닉 상태가 주어진다.
초기 입력 토큰은 문자열-시작 (start-of-string) <SOS> 토큰이고,
첫 은닉 상태는 문맥 벡터(인코더의 마지막 은닉 상태) 이다. (?)



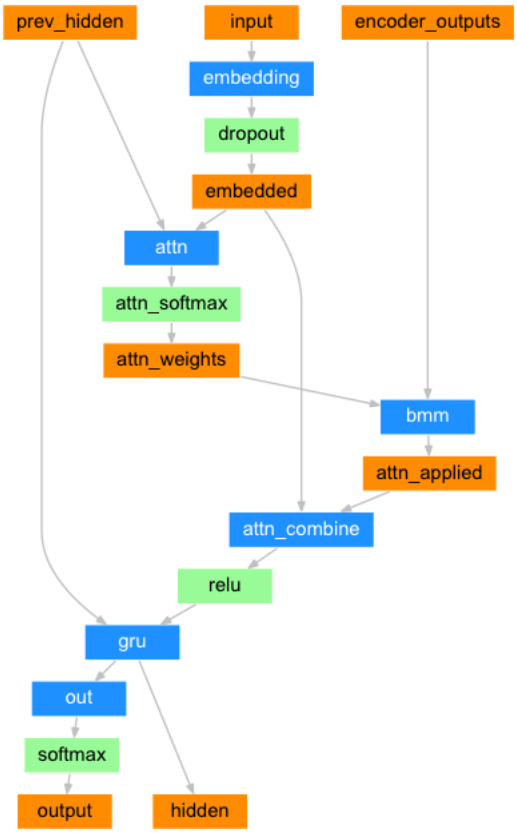
Attention 디코더

문맥 벡터만 인코더와 디코더 사이로 전달 된다면,
단일 벡터가 전체 문장을 인코딩 해야하는 부담을 가지게 된다.

Attention은 디코더 네트워크가 자기 출력의 모든 단계에서 인코더 출력의 다른 부분에 “집중” 할 수 있게 한다. 첫째 Attention 가중치 의 세트를 계산하고 이것을 가중치 조합을 만들기 위해서 인코더 출력 벡터와 곱한다. 그 결과(코드에서 attn_applied)는 입력 시퀀스의 특정 부분에 관한 정보를 포함해야하고 따라서 디코더가 알맞은 출력 단어를 선택하는 것을 도와주게 된다.



어텐션 가중치 계산은 디코더의 입력 및 은닉 상태를 입력으로 사용하는 다른 feed-forwad 계층인 attn 으로 수행된다. 학습 데이터에는 모든 크기의 문장이 있기 때문에 이 계층을 실제로 만들고 학습시키려면 적용 할 수 있는 최대 문장 길이 (인코더 출력을 위한 입력 길이)를 선택해야 한다. 최대 길이의 문장은 모든 Attention 가중치를 사용하지만 더 짧은 문장은 처음 몇 개만 사용한다.



인코더와 디코더를 합친 구조의 학습 단계를 코드로 구현한 것을 아래 더보기에 넣겠다.

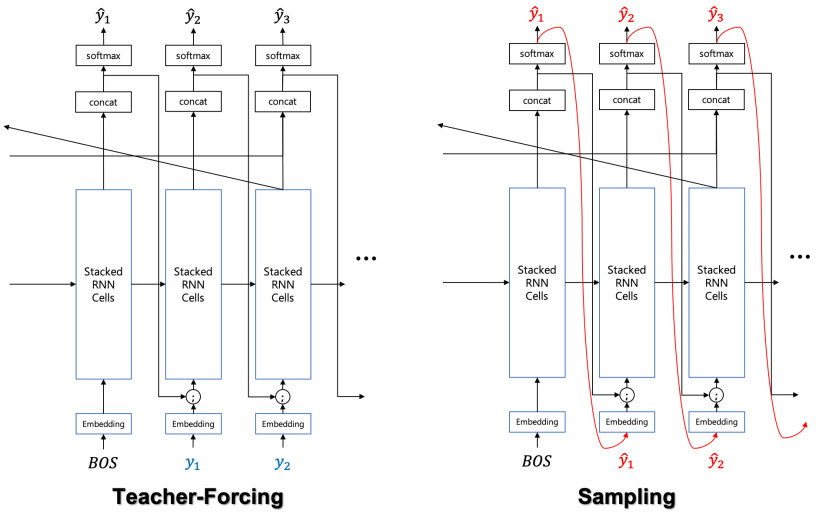
▼ 더보기

```
def train(pairs, source_vocab, target_vocab, encoder, decoder, n_iter, print_every=1000, learning_rate=0.01):
    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()
    # cost function은 NLL(Negative Log Likelihood)를 쓴다. 최종 output이 원래의 단어들과 비교하기 위해 카테고리화 될 것이기 때문에 이 cost functior

    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)
        # hidden state는 맨 처음에는 zero vector를 만들어 넣어 준다.

        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)
            # encoder의 hidden state를 꺼내 온다.
        # encoder의 마지막 hidden state를 decoder의 첫 hidden state으로 넣어주기 위해.

        decoder_input = torch.Tensor([[SOS_token]]).long().to(device) # Start of Sequence Token
```

```
        decoder_hidden = encoder_hidden # 위에서 언급한 과정.

        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 forcing이라고 부르는 방법이다.
            # 다음 GRU에 이전 GRU에서 수행한 연산의 결과가 들어갈 수도 있고, 그냥 정답 값을 넣어줄 수도 있는데, teacher forcing은 후자에 해당한다. 더 빨리

        loss.backward()

        encoder_optimizer.step()
        decoder_optimizer.step()

        loss_iter = loss.item() / target_length

        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))
```

위 코드에서 teacher-forcing이라는 기법이 등장한다.

teacher forcing이란, 훈련 시에는 디코더의 입력으로 이전 time-step의 디코더 출력값이 아닌, 정답 Y 가 들어가는 방식이다. 하지만 추론할 때는 정답 Y 를 모르기 때문에, 이전 time-step에서 계산되어 나온 Y햇_t-1 를 디코더의 입력으로 사용한다.



이 방법은 빨리 수렴한다는 장점이 있지만, 그 대신 network의 학습이 불안해 질 수 있다는 단점도 존재한다. 따라서 random함수를 걸어서 50%, 또는 30%의 확률로 teacher forcing을 하도록 설계하는 방법을 쓸 수도 있다.

시퀀스 데이터(Sequence Data)

https://simonjisu.github.io/nlp/2018/07/05/packedsequence.html

Padding

자연어 처리 NLP, audio data 등의 분야에서는 단어의 길이나 개수 등이 정해져 있지 않아
데이터를 처리하기 곤란한 경우가 있다.

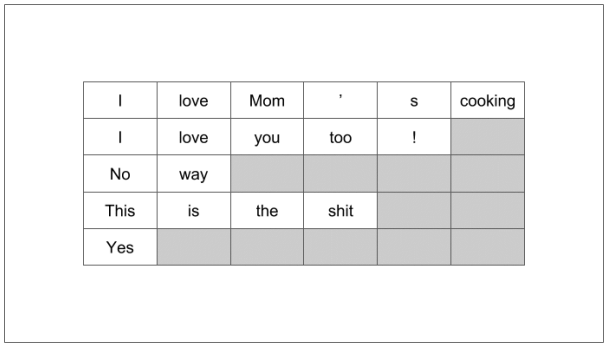
따라서 텐서 연산을 위해 매 배치(batch)마다 Padding을 해주어 문장의 길이를 고정시킨다.

그런데 Padding은 가장 크기가 큰 data에 맞춰서 남는 공간에 <pad>라는 token을 채워넣는 것이므로, batchsize x 최대길이의 크기를 가지는 단일 텐서로 데이터를 표현할 수 있다는 장점이 있지만,연산하지 않아도 되는 부분도 연산해야 한다는 단점이 있다.

Packed-Sequence

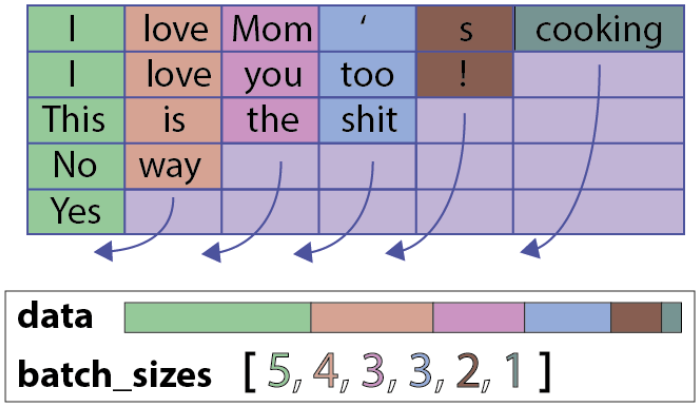
따라서 pad 를 연산 하지 않고 효율적인 진행을 위해 병렬처리를 하려고한다.
그렇다면 다음의 조건을 만족해야한다.

RNN의 히든 스테이트가 이전 타임스텝에 의존해서 최대한 많은 토큰을 병렬적으로 처리해야한다.
각 문장의 마지막 토큰이 마지막 타임스텝에서 계산을 멈춰야한다.



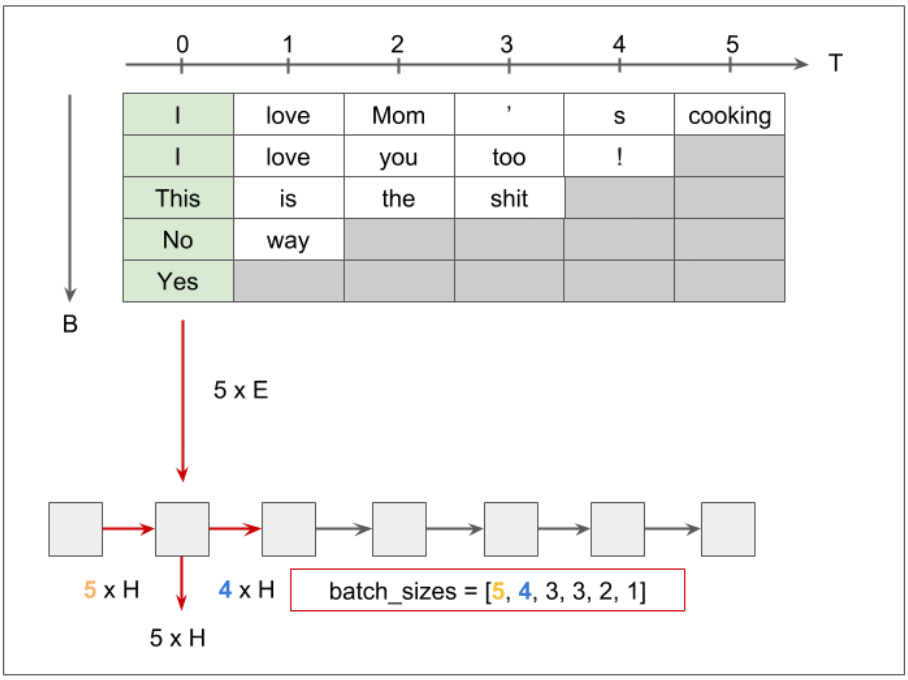
위와 같이 컴퓨터로 하여금
각 **타임스텝**(T=배치내에서 문장의 최대 길이) 마다 일련의 단어를 처리해야한다는 뜻이다.

하지만 T=2,3 인 부분은 중간에 pad가 껴있어 어쩔수 없이 pad를 연산을 하게 되는데,
이를 방지하기 위해서, 아래의 그림같이 각 배치내에 문장의 길이를 기준으로
내림차순으로 **정렬(sorting)**한 후, 하나의 통합된 배치로 만들어주는 것이다.



- **data:** pad 토큰이 제거후 합병된 데이터
- **batch_sizes:** 각 타임스텝 마다 배치를 몇개를 넣는지 기록해 둠

이처럼 Packed-Sequence 는 pad 토큰을 계산하지 않기 때문에 더 빠르게 연산을 처리 할 수 있다.



은닉층에서는 매 타임스텝마다 batch_sizes 를 참고해서
배치수 만큼 은닉층을 골라서 뒤로 전파한다.

기존의 RNN 이라면, **(배치크기 × 문장의 최대 길이 × 층의 갯수)** 만큼 연산을 해야하지만,
(실제 토큰의 갯수 × 층의 갯수) 만큼 계산하면 된다.
이 예시에서는 (5×6×1)=30 → (18×1)=18 로 크게 줄었다.

구현

```
from torch.nn.utils.rnn import pad_sequence, pack_sequence, pack_padded_sequence, pad_packed_sequence
```

torch.nn.utils.rnn으로부터 네 가지 함수를 받아 온다면 일반 Sequence를 packing or padding 해줄 수 있고, Packed Sequence를 Padded Sequence로, 혹은 그 반대로 바꾸어 줄 수도 있다.

PackedSequence and PaddedSequence