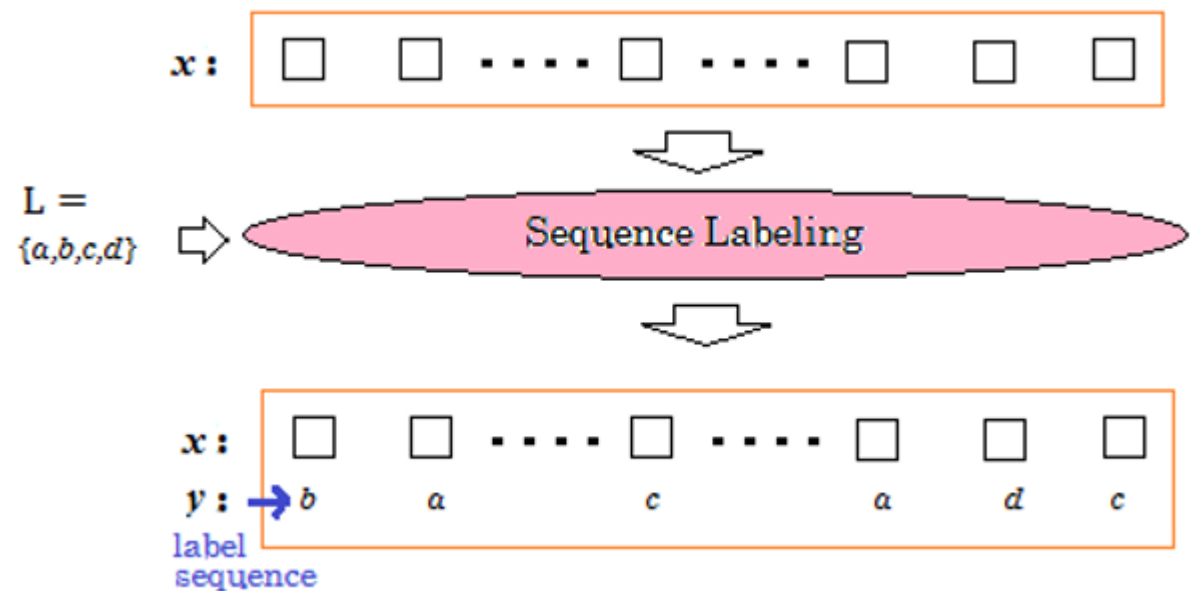


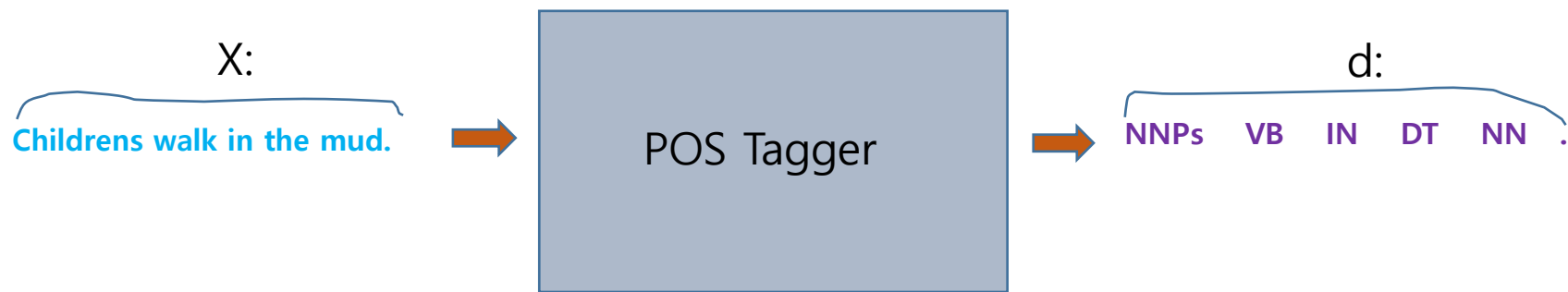
- Classification is done for a sequence of objects simultaneously
- $x = (x_1, \dots, x_T)$: 입력. 1 ~ T 시간대에 걸쳐 각 시간마다 벡터가 입력됨.
- $y = (y_1, \dots, y_T)$: 출력: 각 시간대 마다 입력의 분류 결과인 output label 이 출력됨 (각 시간마다 분류 작업을 수행함.)
- $\Omega = \{l_1, \dots, l_L\}$: class label 집합.



- Sequence Labeling (SL)
 - 여러 시간대에 걸쳐 들어오는 입력에 대하여 각 시간마다 분류 작업을 수행.
 - 입력: a sequence of input data item, $(\mathbf{x}^1, \mathbf{x}^2, \dots, \mathbf{x}^T)$
 - \mathbf{x}^i 는 보통은 벡터임
 - 출력: a sequence of labels, $(\mathbf{y}^1, \mathbf{y}^2, \dots, \mathbf{y}^T)$ where \mathbf{y}^i is a (class) label predicted for \mathbf{x}^i .
 - \mathbf{y}^i 는 클래스 레이블 즉 클래스 번호임
- RNN, LSTM : sequence labeling 에 적합한 기계 학습 모델
 - 시퀀스 내의 모든 시간대의 데이터 아이템에 대하여 동시에(같이) 분류 작업 수행.
 - 즉 다른 시간의 것들도 고려하면서 (서로간의 연관성도 고려하면서) 분류작업을 수행함.
 - 주변문맥(context) 정보의 활용을 가능하게 함.
- SL 의 실제 문제를 살펴보고 이에 LSTM-RNN 모델을 적용하여 보자
 - sequence labeling(SL) 문제에 RNN / LSTM 딥러닝 모델을 적용하여 본다.
 - "영어 품사(part of speech) 태깅" 문제는 전형적인 SL 문제임
 - 이는 자연어처리(natural language processing)의 가장 기초적인 문제임.
 - 딥러닝 라이브러리(Tenforflow/KERAS 또는 Pytorch) 를 이용하여 영어품사태깅 시스템(모델)을 개발해 보자.


품사 (part of speech) 인식(tagging) 문제

- 품사 인식(태깅)이란: 문장 내의 단어 들에 대하여 그 품사를 인식하는 작업
 - 예: "the man can open the lock with a hammer" 내의 각 단어의 품사를 인식하자.
 - 전형적인 인공지능의 문제임: 시퀀스 입력에 대한 분류
- 영어 품사 인식기(tagger) 의 입출력:
 - 입력: 영어 문장 하나(여러 단어를 가짐)
 - 출력: 이 문장 내의 각 단어에 대하여 그것의 품사 번호를 가진 시퀀스(리스트).
- 품사 인식기 개발을 위한 2 가지 접근 방법 :
 - (1) 단순 기법: 각 단어를 놓고 이의 품사를 추정하는 방식으로 Word-wise model 이라 부른다.
 - 이것은 어느 한 시간 만에 대한 입력과 출력을 결정하는 작업이다. 즉 단어 하나를 주고 이의 품사를 알아내는 작업이다.
 - (2) 고급 기법: 한번에 문장 전체의 단어들에 대하여 품사를 결정하고자 한다.
 - 이는 sequence labeling 문제이다.



- A training example = (X, d)
 - X : 한 문장 즉 단어들의 리스트
 - d : 정답 품사열

- 문제에 대한 인공지능 "기계학습" 모델 개발을 위해서는 훈련에 사용할 데이터가 필요하다.
- 이를 위해 1990년대 초에 Penn Tree-Bank를 구축하였다(미국 Pennsylvania University).
- 파일 내의 일부 모습은 아래와 같다: 24개의 디렉토리로 구성, 각 디렉토리에 100개의 파일을 가짐.



```

1
2
3
4 =====
5
6 [ Yields/NNS ]
7 on/IN
8 [ money-market/JJ mutual/JJ funds/NNS ]
9 continued/VBD to/TO slide/VB ,/, amid/IN
10 [ signs/NNS ]
11 that/IN
12 [ portfolio/NN managers/NNS ]
13 expect/VBP
14 [ further/JJ declines/NNS ]
15 in/IN
16 [ interest/NN rates/NNS ]
17 ./
18
19 =====
20
21 [ The/DT average/JJ seven-day/JJ compound/NN yield/NN ]
22 of/IN
23 [ the/DT 400/CD taxable/JJ funds/NNS ]
24 tracked/VBN by/IN
25 [ IBC/NNP 's/POS Money/NNP Fund/NNP Report/NNP ]
26 eased/VBD

```

A red circle highlights the word "funds" in line 8, with a red arrow pointing to it from the label "POS tag".

- 문장에 답(target)을 달아 놓은(tagging) 문장 집단을 annotated corpus (or tagged corpus)라 함
 - 이는 학습 data의 구축에 사용하기 위해 구축함
 - 사람이 수작업으로 준비함 (supervised learning)
- annotated corpus의 예: PennTree bank
 - 영어 자연어처리 작업의 학습용으로 구축함
 - 가장 유명한 데이터 셋
 - 품사태깅, 구문분석 작업을 위함
 - 크기: 약 100만 단어
 - 00 ~ 24 subdirectories이고 각각은 약 100개의 tagged files를 가짐

- 품사 테이블: 모든 품사를 등록한 테이블
- 품사의 번호(index) :
 - 품사마다 고유 번호를 부여함
 - 우리는 1 대신 0 부터 출발시킴
 - 옆의 테이블의 번호에서 1을 뺀 값을 부여함.
- Quote symbols
 - " : straight double quote
 - ": left open double quote
 - ” : right close double quote
 - ' : left open single quote
 - ’ : right close single quote

PRP\$
로 고쳐야 함

Table 2
The Penn Treebank POS tagset.

1. CC	Coordinating conjunction	25. TO	to
2. CD	Cardinal number	26. UH	Interjection
3. DT	Determiner	27. VB	Verb, base form
4. EX	Existential <i>there</i>	28. VBD	Verb, past tense
5. FW	Foreign word	29. VBG	Verb, gerund/present participle
6. IN	Preposition/subordinating conjunction	30. VBN	Verb, past participle
7. JJ	Adjective	31. VBP	Verb, non-3rd ps. sing. present
8. JJR	Adjective, comparative	32. VBZ	Verb, 3rd ps. sing. present
9. JJS	Adjective, superlative	33. WDT	<i>wh</i> -determiner
10. LS	List item marker	34. WP	<i>wh</i> -pronoun
11. MD	Modal	35. WP\$	Possessive <i>wh</i> -pronoun
12. NN	Noun, singular or mass	36. WRB	<i>wh</i> -adverb
13. NNS	Noun, plural	37. #	Pound sign
14. NNP	Proper noun, singular	38. \$	Dollar sign
15. NNPS	Proper noun, plural	39. .	Sentence-final punctuation
16. PDT	Predeterminer	40. ,	Comma
17. POS	Possessive ending	41. :	Colon, semi-colon
18. PRP	Personal pronoun	42. (Left bracket character
19. PP\$	Possessive pronoun	43.)	Right bracket character
20. RB	Adverb	44. "	Straight double quote
21. RBR	Adverb, comparative	45. '	Left open single quote
22. RBS	Adverb, superlative	46. "	Left open double quote
23. RP	Particle	47. '	Right close single quote
24. SYM	Symbol (mathematical or scientific)	48. "	Right close double quote

- 모든 단어를 저장한 테이블을 단어사전이라 부른다: Vocab
 - 시스템이 다루는 문장들에 나올 법한 모든 단어를 가진다:
 - 문제점: 시스템의 사용 중에 단어가 Vocab 에 없는 올 수도 있다
 - 이유: Vocab 는 미리 만들어 놓았는데 시스템의 사용 중에 (즉 testing 중 포함)에는 지금까지 전혀 안 보였던 단어가 나올 수도 있다.
 - 대처법: 이러한 새로운 단어를 "[UNK]" 로 바꾸어 이용한다. "[UNK]" 는 미리 사전에 넣어 놓은 단어이다(단어번호 1).
 - 미리 넣어 놓는 단어로 "[PAD]" 도 있다 (단어번호 0. 이의 필요는 나중에 설명.)
- 사전 만들기
 - Penn tree-bank 에 나타난 모든 문장내의 단어들을 조사한다:
 - penn tree-bank 내의 전체 문장들 (또는 training 용 문장들) 내의 단어들을 조사하여 결정함
 - 출현 횟수가 특정 값 (threshold) 보다 작은 단어들을 Vocab 에서 제거한다. (예: threshold = 3)
 - 결국 이런 제거된 단어들이 나타나면 "[UNK]" 로 간주하게 된다.
 - Threshold 를 조정하면 사전의 크기를 조정할 수 있다.
 - 사전의 크기를 줄이는 기타 추가적인 방법:
 - 모든 수(number) 형태의 모든 단어를 "[NUM]" 이란 특별 단어로 대체함. (주: 우리 과제에서는 사용하지 않음.)
- 단어번호(word index):
 - 단어마다 고유번호를 부여한다. 이를 단어번호(word index)라 부른다.
 - 우리는 시스템 개발에서 단어 대신 단어 번호를 이용한다.
 - 이유: neural network 은 수(number)를 처리하는 모델이므로 단어 대신 단어번호가 더 다루기 편리하다.
 - Vocab 는 python 의 dictionary 데이터구조를 이용하면 편리하다.
- 우리 과제의 사전 크기: Vocab_size = 51,459
 - Penn treebank 의 전체 코퍼스(문장집합)를 대상으로 사전을 만드는 경우.

word	word-index
[PAD]	0
[UNK]	1
.	
.	
million	26
with	27
Mr.	28
was	29
be	30
are	31
its	32
n't	33
has	34
an	35
have	36
will	37
he	38
or	39
company	40
which	41
would	42
year	43
--	44
market	45
about	46
were	47
they	48
says	49
this	50
more	51
had	52
.	
.	

작업 1: 문장마다 단어열과 품사열을 준비

- 원본 훈련 데이터를 읽어서 한 문장의 정보를 한 줄에 준비한다.
 - 문장의 단어마다 '/' 글자를 바로 뒤에 붙이고 그 바로 뒤에 품사명을 붙인다.
 - 여기서 품사명은 단어의 정답(target) 품사명이다.
 - 파일명: all_word_pos_sentences_all.txt

```

3
4 =====
5
6 [ Yields/NNS ]
7 on/IN
8 [ money-market/JJ mutual/JJ funds/NNS ]
9 continued/VBD to/TO slide/VB ,/, amid/IN
10 [ signs/NNS ]
11 that/IN
12 [ portfolio/NN managers/NNS ]
13 expect/VBP
14 [ further/JJ declines/NNS ]
15 in/IN
16 [ interest/NN rates/NNS ]
17 ./
18
19 =====
20
21 [ The/DT average/JJ seven-day/JJ compound/NN yield/NN ]
22 of/IN
23 [ the/DT 400/CD taxable/JJ funds/NNS ]
24 tracked/VBN by/IN
25 [ IBC/NNP 's/POS Money/NNP Fund/NNP Report/NNP ]
26 eased/VBD

```

Dr./NNP Talcott/NNP led/VBD a/DT team/NN of/IN researchers/NNS from/IN the/DT National/NNP Cancer/NNP Institute/NNP and/CC the/DT Lorillard/NNP spokeswoman/NN said/VBD asbestos/NN was/VBD used/VBN in/IN ``/`` very/RB modest/JJ amounts/NNS ``/`` in/IN I/IN From/IN 1953/CD to/TO 1955/CD ,/, 9.8/CD billion/CD Kent/NNP cigarettes/NNS with/IN the/DT filters/NNS were/VBD sold/VBN ,/, t/IN Among/IN 33/CD men/NNS who/WP worked/VBD closely/RB with/IN the/DT substance/NN ,/, 28/CD have/VBP died/VBN --/: more/JJ than/IN three/CD times/NNS the/DT expected/VBN number/NN ./.

Four/CD of/IN the/DT five/CD surviving/VBG workers/NNS have/VBP asbestos-related/JJ diseases/NNS ,/, including/VBG three/CD wi/IN The/DT total/NN of/IN 18/CD deaths/NNS from/IN malignant/JJ mesothelioma/NN ,/, lung/NN cancer/NN and/CC asbestosis/NN was/VBD ``/`` The/DT morbidity/NN rate/NN is/VBZ a/DT striking/JJ finding/NN among/IN those/DT of/IN us/PRP who/WP study/VBP asbestos-related/JJ diseases/NNS ,/, ``/`` said/VBD Dr./NNP Talcott/NNP ./.

The/DT percentage/NN of/IN lung/NN cancer/NN deaths/NNS among/IN the/DT workers/NNS at/IN the/DT West/NNP Groton/NNP ,/, Mass./IN The/DT plant/NN ,/, which/WDT is/VBZ owned/VBN by/IN Hollingsworth/NNP &/CC Vose/NNP Co./NNP ,/, was/VBD under/IN contract/NN The/DT finding/NN probably/RB will/MD support/VB those/DT who/WP argue/VBP that/IN the/DT U.S./NNP should/MD regulate/VB the/DT class/NN of/IN asbestos/NN including/VBG crocidolite/NN The/DT U.S./NNP is/VBZ one/CD of/IN the/DT few/JJ industrialized/VBN nations/NNS that/WDT does/VBZ n't/RB have/VB a/DT higher/JJR standard/NN of/IN regulation/NN for/IN the/DT smooth/JJ ,/, needle-like/JJ fi that/WDT are/VBP classified/VBN as/IN amphiboles/NNS ,/, according/VBG to/TO Brooke/NNP T./NNP Mossman/NNP ,/, a/DT professor/ More/RBR common/JJ chrysotile/NN fibers/NNS are/VBP curly/JJ and/CC are/VBP more/RBR easily/RB rejected/VBN by/IN the/DT body/ In/IN July/NNP ,/, the/DT Environmental/NNP Protection/NNP Agency/NNP imposed/VBD a/DT gradual/JJ ban/NN on/IN virtually/RB al By/IN 1997/CD ,/, almost/RB all/DT remaining/VBG uses/NNS of/IN cancer-causing/JJ asbestos/NN will/MD be/VB outlawed/VBN ./.

About/IN 160/CD workers/NNS at/IN a/DT factory/NN that/WDT made/VBD paper/NN for/IN the/DT Kent/NNP filters/NNS were/VBD exposed/VBN to/TO asbestos/NN in/IN the/DT 1950s/CD ./.

작업 2: 단어는 단어번호, 품사명은 품사 번호로 나타낸 파일

- 작업1 에서 준비한 파일로 부터 단어는 단어번호로 품사명은 품사번호로 표시한 파일을 준비한다.
- 그리고 한 문장마다 3 줄을 준비한다.
 - 첫째 줄: 문장 내 단어들의 단어 번호 열 (단어 번호: 앞에서 구축한 단어 사전 이용.)
 - 둘째 줄: 문장 내 단어들의 품사 인덱스 열(품사 인덱스: 미리 준비한 품사 사전 이용.)
 - 그 뒤에 빈 줄이 나온다.
- 결과 파일명: all_index_sentences_all.txt

12555	27961	2	3831	82	492	2	37	2430	3	353	25	7	12554	319	446	1435	4
13	13	39	1	12	6	39	10	26	2	11	5	2	6	11	13	1	38
28	27961	16	195	5	51458	6341	2	3	3314	2244	136	4					
13	13	31	11	5	13	13	39	2	13	28	11	38					
9540	20493	2	1782	82	492	8	389	195	5	5542	3383	16684	995	2	29	552	7
13	13	39	1	12	6	0	6	11	5	13	13	13	13	39	27	29	2
98	1102	5	3731	662	291	6	148	7362	5133	7811	34	1008	7	282	836	5	147
2	11	5	11	19	29	24	26	13	11	12	31	29	2	6	11	5	11
17	3731	10317	2	14277	2	16	4784	14276	662	21	8863	3	16683	2	27	144	224
2	11	11	39	11	39	31	19	6	5	17	31	2	12	39	5	19	6

- all_index_sentences_all.txt 은 PTB 의 모든 문장에 대한 것이다.
 - 이것을 3 부분으로 나눈다.
 - 훈련(train), 검증(validation), 테스트(test) 로 나누어 준비함
 - 전체를 적절한 비율로 배분함 (예: 80:10:10)
 - 파일명: all_index_sentences_train.txt , all_index_sentences_validation.txt , all_index_sentences_test.txt
- 프로그램이 사용할 훈련예제 집합 : 3 가지 정보를 준비.
 - X (예제의 입력아이템) 의 리스트, Y(예제의 정답)의 리스트, 예제(문장) 길이(단어수)의 리스트
list_X, list_Y, list_leng
 - Padding and truncation:
 - 예제들은 batch 형태로 모델에 제공되어야 한다. 따라서 모든 문장의 길이를 같은 특정길이(Max_sequence_length; MSL; T)로 맞춰 주어야 한다.
 - 미리 설정한 특정길이와 같게 맞춰 주어야 함 (이 길이를 T 이라 하자; (T=MSL)
 - 어느 문장의 길이 (단어수)가 T 보다 작으면 모자라는 단어수 만큼 padding 용 단어 "[PAD]" (번호 0) 으로 채워 줌 (Padding 작업).
 - 어느 문장의 길이가 T 보다 크면 T 에서 잘라 준다. (Truncation 작업).
- 적용분야(train, val.,test) 별로 전체 문장 정보를 가진 파일 "all_index_sentences_???.txt" 을 읽어서 위의 3 개의 리스트를 준비함
 - ??? 자리에 train, validation, test 가 들어감
- 이는 함수 load_X_and_Y 를 이용함.
 - 이 함수는 화일명을 입력받아서,
 - 읽기 작업을 하여 위 3 리스트를 반환함: x_train, y_train, x_validation, y_validation, ...

- 단어를 번호 하나 즉 정수 하나로 나타내는 것은 너무 단순하다.
 - 단어의 의미를 나타내기 어렵다
- 그래서 단어를 일정한 크기(예: d_emb = 100) 의 실수 vector 로 나타내기로 한다.



- 이를 word embedding vector 라 부른다.
 - 단어 마다 이 벡터를 둔다.
 - 이 벡터의 각 원소의 값은 훈련을 통해 구한다.
 - 즉 단어 벡터의 각 원소를 parameter 로 간주한다.
-
- 결국 단어사전에 대응하여 word embedding matrix 가 존재한다.

index	word
0	apple
1	box

index	word embedding vector
0	(1.24, 2.56, -1.89,)
1	(0.51, -3.17, 9.34,)

- Embedding layer의 역할:

- (1) word embedding matrix 생성
- (2) RNN(or LSTM) 으로 입력되는 단어열에 대한 word vector 공급.



- (1) Word embedding matrix 생성

- Pytorch 의 nn.Embedding 클래스를 이용함.

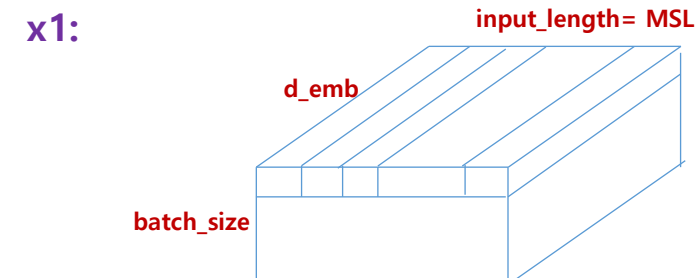
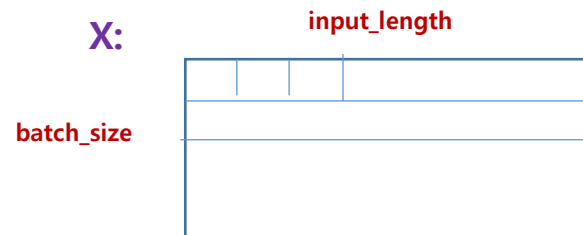
```
self.embedding = nn.Embedding(token_vocab_size, dim_embedding, padding_idx=0)
```

- 위와 같이 호출하면 객체 self.embedding 이 생기면서 이 안에 그림과 같이 word embedding matrix 가 생성된다.

- (2) 입력 단어열에 대한 word vector 들을 가져오기:

- 위의 self.embedding 객체를 호출하여 가져 올 수 있다: X: 입력단어(id)열, x1: 단어id 마다 해당 word vector 로 변경.

```
x1 = self.embedding(X) # output shape is (batch, msl, dim_embedding)
```



- RNN or LSTM 층의 구현 :

- 클래스 nn.RNN or nn.LSTM 을 이용한다:

```
## self.lstm = nn.LSTM(input_size=dim_embedding, hidden_size=hidden_state_size, #
self.lstm = nn.RNN(input_size=dim_embedding, hidden_size=hidden_state_size, #
                    num_layers=num_hidden_layers, batch_first=True, bidirectional=True)
```

- 층을 구현하는 객체 self.lstm 이 생성된다.

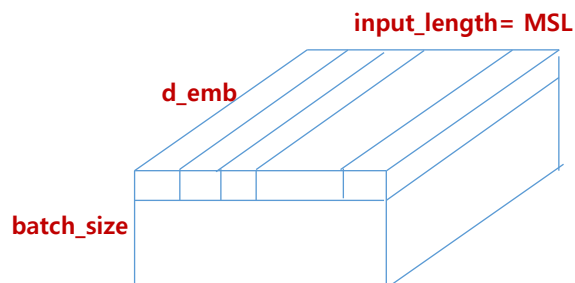
- RNN or LSTM 층을 호출하여 이용하기:

- 입력단어열에 대한 word vector 열, x1, 을 RNN or LSTM 층에 입력하여 결과를 얻으려면 층 객체를 호출하여야 한다.

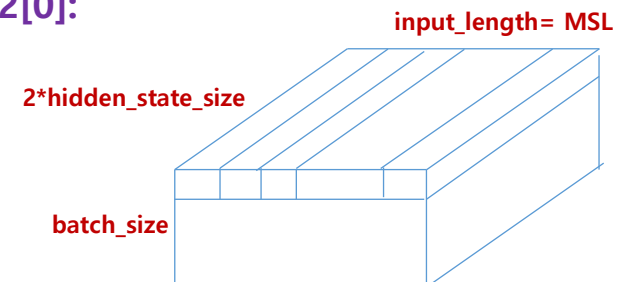
```
x2 = self.lstm(x1) # output shape is (batch, msl, 2*hidden_state_size)
x2 = x2[0] # the seq of the outputs of all timesteps.
```

- Self.lstm 의 출력은 3 가지 정보를 출력한다. 첫번째 것(x2[0])이 우리가 생각하는 각 시간의 최종층 hidden_state이다.

x1:



x2[0]:



```
17 import numpy as np
18 import torch
19 from torch import nn
20 from torch.utils.data import Dataset
21 import time
22 from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler
23 from torch import optim
24
25 from google.colab import drive
26 drive.mount('/content/drive')
27
28 # %cd /content/drive/MyDrive
29 print("current working directory=")
30 !pwd
31
32
33 device = torch.device("cpu")
34 print("device= ", device)
35
36 MSL = 64                # maximum sequence length of input
37 NUM_CLASS = 50          # 영어 품사 종류 개수([PAD], [UNK] 포함)
38 SZ_WORD_VOCAB = 51423   # size of vocabulary of words
39 DIM_EMBEDDING = 100     # dimension size of word embedding vectors
40 NUM_HIDDEN_LAYERS = 2    # number of hidden layers of LSTM or RNN
41 SZ_HIDDEN_STATE = 256   # number of neurons of a hidden layer
42 num_EPOCHS = 10         # number of epochs in training
43 BATCH_SIZE = 4          # 배치 크기
```

```
def load_X_and_Y(path_index_file, num_line_to_read):
    fp= open(path_index_file, "r", encoding="utf-8")
    list_X = []
    list_Y = []
    list_leng = []
    line_cnt = 0

    while True:
        # read two lines
        wordline = fp.readline()
        line_leng = len(wordline)

        if line_leng == 0:
            break # end of file has come.
        if line_leng == 1:
            continue # empty line used as sentence delimiter

        # The line read just before is a line of word indices.
        # The next line should be the corresponding pos index line.
        posline = fp.readline()
        w_index = wordline.split()
        p_index = posline.split()
        line_cnt += 1
```

```
# X : a list of indices of words in a sentence.
# Y : a list of pos indices of words in the sentence of X.
X = []
Y = []

leng = len(w_index)
if leng > MSL-1:
    leng = MSL-1 # 길이를 줄여서 truncation 을 수행하는 효과를 얻는다.

for i in range(leng):
    X.append(word_index)
    Y.append(int(p_index[i]))

# padding 이 필요하면 padding 을 수행한다. 위에서 MSL-1을 최대로 해서
# 최소 한 개의 pad 넣는다.
if leng < MSL:
    for i in range(leng, MSL):
        X.append(0) # word index of '[PAD]' which is 0 is added.
        Y.append(0) # pos index of '[PAD]' which is 0 is added.

list_X.append(X)
list_Y.append(Y)
list_leng.append(leng)

if line_cnt >= num_line_to_read:
    break
fp.close()
return list_X, list_Y, list_leng
```

```

104 # 훈련 예제 리스트를 준비한다.
105 print("reading train data.")
106 x_train, y_train, leng_train = load_X_and_Y("./English_POS_tagging_data/all_index_sentences_train.txt", 20000)
107
108 print("num of sentences=", len(x_train), len(y_train), len(leng_train))
109 print(x_train[0])
110
111 train_X = torch.LongTensor(x_train)
112 train_Y = torch.LongTensor(y_train)
113 train_Leng = torch.IntTensor(leng_train)
114
115 # test 예제 리스트를 준비한다.
116 print("reading test data.")
117 x_test, y_test, leng_test = load_X_and_Y("./English_POS_tagging_data/all_index_sentences_test.txt", 2000)
118 print("reading done.")
119 #time.sleep(200)
120 test_X = torch.LongTensor(x_test)
121 test_Y = torch.LongTensor(y_test)
122 test_Leng = torch.IntTensor(leng_test)
    
```



```

124 # 모델 설계:
125 # 여러 층의 LSTM 위에 3개의 FF 층을 올린다.
126 class POS_model(nn.Module):
127     def __init__(self, token_vocab_size, dim_embedding, num_hidden_layers, hidden_state_size):
128         super(POS_model, self).__init__()
129         self.embedding = nn.Embedding(token_vocab_size, dim_embedding, padding_idx=0)
130
131         ##self.lstm = nn.RNN(input_size=dim_embedding, hidden_size=hidden_state_size, \ <-- RNN 사용에 이용.
132         self.lstm = nn.LSTM(input_size=dim_embedding, hidden_size=hidden_state_size, \
133                             num_layers=num_hidden_layers, batch_first=True, bidirectional=True)
134
135         self.linear1 = nn.Linear(2*hidden_state_size, 512, bias=True)
136         self.relu = nn.ReLU()
137
138         self.linear2 = nn.Linear(512, 256, bias=True)
139         self.relu = nn.ReLU()
140
141         self.linear3 = nn.Linear(256, NUM_CLASS) # NUM_CLASS is the number of classes of POS
142
143     def forward(self, X):
144         # X : shape (batch, msl) where each example is a list of token ids of length msl.
145         x1 = self.embedding(X) # output shape is (batch, msl, dim_embedding)
146
147         x2 = self.lstm(x1) # LSTM은 3 가지 텐서를 출력한다.
148         x2 = x2[0] # 이 들중 첫 번째가 우리가 원하는 것임: (batch, MSL, hidden_state)
149
150         x3 = self.linear1(x2) # FF층 1: output shape is (batch, msl, 512).
151         x4 = self.relu(x3)
152         x5 = self.linear2(x4) # FF층 2:output shape is (batch, msl, 256).
153         x6 = self.relu(x5)
154         x7 = self.linear3(x6) # FF층 3:output shape is (batch, msl, num_class).
155         return x7
156
157 model = POS_model(SZ_WORD_VOCAB, DIM_EMBEDDING, NUM_HIDDEN_LAYERS, SZ_HIDDEN_STATE)

```

- 한 학습예제는 (X, Y) 형태이다.
 - X: 한 문장의 단어 sequence(단어열) 이다. (각 값은 id)
 - Y: X 의 각 토큰에 대하여 POS-레이블로 구성된 label sequence 이다. (각 값은 id)
 - 모델에게는 텐서 타입의 batch 를 만들어 제공하여야 한다.
 - 따라서 모든 예제가 동일한 shape 을 가져야 한다.
 - 결국, token sequence 의 길이를 특정길이(MSL) 로 통일하여야 한다.
 - 짧은 문장은 padding 이 필요하고,
 - 너무 긴 문장은 truncation 이 필요한 이유이다.

- model 에게 학습데이터를 batch 로 제공한다.
- 한 batch 의 shape: (BATCH_SIZE, MSL)
- data type: LongTensor
- 우리는 전체 예제들을 담은 3개의 리스트를 준비한다:
 - train_X : X 들의 리스트
 - train_Y : 위의 각 X 에 대응하는 Y 의 리스트
 - Train_Leng: X의 실제 문장의 길이의 리스트
- 이를 batch 별로 모델에 제공하는 작업은 pytorch 의 DataLoader 를 이용한다.

- 데이터의 tensor 화, batch loader 준비

```
159 train_data = TensorDataset(train_X, train_Y, train_Leng)
160 train_sampler = RandomSampler(train_data)
161 train_dataloader = DataLoader(train_data, sampler=train_sampler, batch_size=BATCH_SIZE, drop_last=True)
162
163 test_data = TensorDataset(test_X, test_Y, test_Leng)
164 test_sampler = RandomSampler(test_data)
165 test_dataloader = DataLoader(test_data, sampler=test_sampler, batch_size=BATCH_SIZE, drop_last=True)
```

- Batch loader 의 이용방법

```
185 for i, batch in enumerate(train_dataloader):
186
187     batch = tuple(r.to(device) for r in batch)
188     X, Y, Leng = batch
189
190     optimizer.zero_grad()
191     model.zero_grad()
192
193     # shape of input to model : (batch, MSL)
194     # shape of output from model : (batch, MSL, num_class)
195     preds = model(X)
```

```
#optimizer = optim.SGD(model.parameters(), lr=1e-5)
#optimizer = optim.Adam(model.parameters(), lr=1e-5, weight_decay=0.1)
#optimizer = optim.AdamW(model.parameters(), lr=1e-3, betas=(0.9, .999),\
                        #eps=1e-08, weight_decay=0.01)

optimizer = optim.AdamW(model.parameters(), lr=1e-4, betas=(0.9, .999),\
                        eps=1e-08, weight_decay=0.01)
```

```
CLASS torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100,  
    reduce=None, reduction='mean')
```

Parameters

- **ignore_index** (*int, optional*) – Specifies a target value that is ignored and does not contribute to the input gradient. When `size_average` is `True`, the loss is averaged over non-ignored targets.
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`.
`'none'` : no reduction will be applied, `'mean'` : the weighted mean of the output is taken, `'sum'` : the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

The *input* is expected to contain raw, unnormalized scores for each class.

input has to be a Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, \dots, d_K)$ with $K \geq 1$ for the K -dimensional case (described later).

- Input: (N, C) where $C = \text{number of classes}$, or $(N, C, d_1, d_2, \dots, d_K)$ with $K \geq 1$ in the case of K -dimensional loss.
- Target: (N) where each value is $0 \leq \text{targets}[i] \leq C - 1$, or $(N, d_1, d_2, \dots, d_K)$ with $K \geq 1$ in the case of K -dimensional loss.
- Output: scalar. If `reduction` is `'none'`, then the same size as the target: (N) , or $(N, d_1, d_2, \dots, d_K)$ with $K \geq 1$ in the case of K -dimensional loss.

- LSTM 은 RNN 의 일종으로서 각 timestep 에서의 loss 의 합으로 전체 loss 를 구한다.
- 특정 timestep (예를 들어 t) 에서의 loss 는 입력 token 에 대한 label prediction 과 target label에 의한 cross-entropy 이다.
 - 최종층은 POS_label 의 총 갯수인 num_class 개의 뉴론으로 구성됨.
 - 각 뉴론 i 는 레이블 i 를 지지하는 정도를 나타내는 출력을 내준다.
(우리 모델은 softmax 층을 적용하지 않은 값을 출력함.)
 - t 에서의 target label (index) 가 주어진다. 이 값이 d 라고 하자.
- Cross-entropy at timestep t = $-\ln(y_d^t)$ where y_d^t is the softmax result for neuron d at time t.
- 우리는 다음 pytorch 함수를 이용하여 loss 를 계산한다.

```
172 # idx 0 of [PAD] label in target is ignored in computing the loss.
173 loss_fn = torch.nn.CrossEntropyLoss(ignore_index=0, reduction='mean')
```

- ignore_index: 무시할 target label. padding 할 때 0 을 target label 로 공급하였음. 이들은 loss 계산에 참여시키지 않음.
- 이 함수의 특징은 softmax 를 적용하지 않은 최종층 출력을 이용한다.
- 이 함수로의 입력 데이터는 (batch, class-label, timestep)의 shape 이어야 함.
- 우리 모델의 출력의 shape 과 순서가 다름.
 - transpose 가 필요한 이유임

```
preds = model(X)
preds_tr = torch.transpose(preds, 1, 2)
loss = loss_fn(preds_tr, Y)
```

- backward pass : loss 텐서의 backward 메소드의 호출
 - 모든 parameter 들의 gradient 계산
- parameter update: optimizer 객체의 메소트 step 의 호출
 - 모든 parameter 들이 새로운 값으로 갱신됨

```
loss.backward()
```

```
optimizer.step()
```



```

180 for e in range(num_EPOCHS):
181     #if e > 0:
182         #break
183     model.train()
184     total_loss = 0.0
185     for i, batch in enumerate(train_dataloader):
186
187         batch = tuple(r.to(device) for r in batch)
188         X, Y, Leng = batch
189
190         optimizer.zero_grad()
191         model.zero_grad()
192
193         # shape of input to model : (batch, MSL)
194         # shape of output from model : (batch, MSL, num_class)
195         preds = model(X)
196
197         # Transpose is needed since crossentropyloss requires a shape of (N, C, MSL),
198         # where N:batch, C:category, MSL: seq length.
199         preds_tr = torch.transpose(preds, 1, 2)
200
201         # shape of Y should be (batch, msl).
202         loss = loss_fn(preds_tr, Y)
203         loss.backward()
204         optimizer.step()
205         total_loss += loss.item()
206
207     avg_loss = total_loss / total_num_batches
208     print("Epoch: ", e, " has finished. Avg_loss= ", avg_loss)

```

필요한 이유는?

- training 의 각 epoch 를 마친 후에 test data 를 이용하여 검증을 수행한다.
 - 이유는 loss 보다는 accuracy 를 계산해 보는 것이 보다 더 성능 확인에 도움이 된다.

```
209
210 ##### Validation after one epoch #####
211 total_word_cnt = 0 # 모든 문장들에서 패드가 아닌 총 단어 수
212 total_success_cnt = 0 # 모든 문장에서 패드가 아닌 단어 중 품사를 맞춘 단어 수
213 model.eval()
214 with torch.no_grad():
215     print("Validation starts.")
216     for k, batch in enumerate(test_dataloader):
217
218         batch = tuple(r.to(device) for r in batch)
219         X, Y, Leng = batch
220
221         preds = model(X) # 배치에 대한 모델의 출력. shape=(batch, MSL, num_class)
222
223         pred_label_batch = torch.argmax(preds, dim=2) # 각 시간마다 최대 확률인 class 를 알아 냄.
224
225         for i in range(len(Y)):
226             # 배치 내의 예제 i 에 대하여 각 단어의 맞춤 여부를 확인한다.
227             target_label_seq = Y[i] # 한 문장에 대한 정답 레이블들
228             pred_label_seq = pred_label_batch[i] # 한 문장에 대한 단어들의 예측된 레이블들
229             leng = Leng[i].item()
230             match_cnt = 0 # 문장 내의 맞춘 단어 수 초기화.
231             for j in range(leng):
232                 # 각 단어마다 품사 예측의 성공 여부를 카운트한다.
233                 if pred_label_seq[j] == target_label_seq[j]:
234                     match_cnt += 1
235
236             total_word_cnt += leng
237             total_success_cnt += match_cnt
238         vaccuracy = float(total_success_cnt) / float(total_word_cnt)
239         print("\nValidation accuracy after epoch ", e, " = ", vaccuracy)
240 #####
```