NPL
WITH PYTORCH

PART I

TORCHTEXT

TORCHTEXT

◆ PyTorch 텍스트 라이브러리

- 텍스트에 대한 추상화 기능 제공하는 **자연어 처리 라이브러리**
- **데이터 처리 유틸리티와 인기 있는 자연어 데이터 세트로 구성**
- **2024년 4월 v0.18.0 마지막 안정적인 릴리스 버전 ➔ 개발 중단**

- 제공 기능
  - 파일 로드(File Loading) : 다양한 포맷 코퍼스 로드
  - 토큰화(Tokenization) : 문장을 단어 단위로 분리
  - 단어 집합(Vocab) : 단어 집합 생성
  - 정수 인코딩(Integer encoding) : 전체 코퍼스 단어들 고유한 정수 맵핑
  - 단어 벡터(Word Vector) : 단어 집합 단어들에 고유한 임베딩 벡터 생성
  - 배치화(Batching) : 훈련 샘플들 배치 생성 및 패딩 작업(Padding) 진행

TORCHTEXT

◆ 버전 및 설치

- https://pypi.org/project/torchtext/

PyTorch version	torchtext version	Supported Python version
nightly build	main	>=3.8, <=3.11
2.2.0	0.17.0	>=3.8, <=3.11
2.1.0	0.16.0	>=3.8, <=3.11
2.0.0	0.15.0	>=3.8, <=3.11
1.13.0	0.14.0	>=3.7, <=3.10
1.12.0	0.13.0	>=3.7, <=3.10
1.11.0	0.12.0	>=3.6, <=3.9
1.10.0	0.11.0	>=3.6, <=3.9
1.9.1	0.10.1	>=3.6, <=3.9
1.9	0.10	>=3.6, <=3.9

TORCHTEXT

◆ 버전 및 설치

- **torchtext v0.11.0**

[가상환경]

```
conda create -n TEXT_011_110_38 python=3.8
conda env list
```

[파이토치]

```
conda install pytorch==1.10.0 torchvision==0.11.0 torchaudio==0.10.0 cpuonly -c pytorch
```

[토치텍스트]

```
conda install -c pytorch torchtext==0.11.0
```

TORCHTEXT

◆ 버전 및 설치

- **torchtext v0.15.0**

[가상환경]

```
conda create -n TEXT_015_200_38 python=3.8
conda env list
```

[파이토치]

```
conda install pytorch==2.0.0 torchvision==0.15.0 torchaudio==2.0.0 cpuonly -c pytorch
```

[토치텍스트]

```
conda install -c pytorch torchtext==0.15.0
```

TORCHTEXT

◆ 버전 및 설치

- **torchtext v0.18.0** ← 최종 마지막 버전

[가상환경]

```
conda create -n TEXT_018_230_38 python=3.8
```

[파이토치]

```
conda install pytorch torchvision torchaudio cpuonly -c pytorch
```

[토치텍스트- 의존성패키지]

```
conda install -c conda-forge portalocker>=2.0.0
```

[토치텍스트]

```
conda install -c pytorch torchtext==0.18.0 torchdata
```

TORCHTEXT

◆ 서브 패키지

- torchtext.nn
- torchtext.data.functional
- torchtext.data.metrics
- torchtext.data.utils
- torchtext.datasets
- torchtext.vocab
- torchtext.utils
- torchtext.transforms
- torchtext.functional
- torchtext.models

torchtext.data.functional

- generate_sp_model
- load_sp_model
- sentencepiece_numericalizer
- sentencepiece_tokenizer
- custom_replace
- simple_space_split
- numericalize_tokens_from_iterator
- filter_wikipedia_xml
- to_map_style_dataset

torchtext.vocab

- Vocab
- vocab
- build_vocab_from_iterator
- Vectors
- GloVe
- FastText
- CharNGram

TORCHTEXT

◆ 서브 패키지

Docs > torchtext.datasets

Datasets

- Text Classification
  - AG_NEWS
  - AmazonReviewFull
  - AmazonReviewPolarity
  - CoLA
  - DBpedia
  - IMDb
  - MNLI
  - MRPC
  - QNLI
  - QQP
  - RTE
  - SogouNews
  - SST2
  - STSB
  - WNLI
  - YahooAnswers
  - YelpReviewFull
  - YelpReviewPolarity

Docs > torchtext.datasets

- Language Modeling
  - PennTreebank
  - WikiText-2
  - WikiText103
- Machine Translation
  - IWSLT2016
  - IWSLT2017
  - Multi30k
- Sequence Tagging
  - CoNLL2000Chunking
  - UDPOS
- Question Answer
  - SQuAD 1.0
  - SQuAD 2.0
- Unsupervised Learning
  - CC100
  - EnWik9

TORCHTEXT

◆ iterator 반복자

- **순차적으로 다음 데이터를 리턴할 수 있는 객체**
- **next() 함수 내장**해서 순환하는 다음 값 반환

- 생성 ➔ iter( ) 함수

```
a = [1, 2, 3]
iter_a = iter( a )
print( type(iter_a) )
```

TORCHTEXT

◆ iterator 반복자

```
class MyIter:
    def __init__(self, data):
        self.data = data
        self.posiion=0

    def __iter__(self) : return self

    def __next__(self):
        if self.posiion >= len(self.data): raise StopIteration
        result = self.data[self.posiion]
        self.posiion += 1
        return result
```

_ _iter_ _ 메서드 : 반복자 반환

_ _next_ _ 메서드 : 다음 요소 반환

TORCHTEXT

◆ iterator 반복자

```
class ReverseIter:
    def __init__(self, data):
        self.data = data
        self.posiion= len(self.data) - 1

    def __iter__(self) : return self

    def __next__(self) :
        if self.posiion >= len(self.data): raise StopIteration
        result = self.data[self.posiion]
        self.posiion -= 1
        return result
```

_ _iter_ _ 메서드 : 반복자 반환

_ _next_ _ 메서드 : 다음 요소 반환

TORCHTEXT

◆ iterator 반복자

```
class MyCounter:
    def __init__(self, data):
        self.data = data

    def __iter__(self):
        return MyIter(self.data)
```

* _ _iter_ _ 메서드 존재하는 iterable 객체

* Iterator 객체 반환함!

TORCHTEXT

◆ Generator 반복자

- **이터레이터를 생성해 주는 함수**
- 함수 내부에 **yield** 사용되며, yield로 호출한 곳에 값 전달
- 호출 시에 값을 메모리에 올림 ➔ **지연 평가(Lazy Evaluation) 방식/메모리 효율적**
- 내부 : _ _iter_ _(), _ _next_ _() 존재

- **키워드 yield**
  - **잠시 함수 실행 멈춤 ➔ 호출한 곳에 값 전달**
  - 현재 실행 상태 계속 유지/ 다시 함수 호출 시 현재 실행 상태 기반 코드 실행
  - return처럼 **값 반환 후 종료 되지 않음!**

TORCHTEXT

◆ Generator 반복자

- Generator Expression 또는 Generator Comprehension

  * 형식 : (express for in)

```
square_gen = ( num ** 2 for num in range(5) )

print( type(square_gen) )

next(square_gen), next(square_gen), next(square_gen)
```

TORCHTEXT

◆ Generator 반복자

```
def generator_func():
    for i in [11,22,33]:
        yield i
```

```
gen=generator_func()
print(f'get => {gen}')
```

get => <generator object generator_func at 0x0000021FB53F3F20>

```
def generator_func():
    a = [11,22,33]
    yield from a
```

```
for value in gen:
    print( value )
```

1 2 3

TORCHTEXT

◆ 필드 정의 - torchtext.legacy.data.Field

❖ 피쳐별 전처리 진행 방법 지정 ➔ torchtext v0.10.0, 0.11.0

```
# 필드 정의
TEXT = data.Field(sequential=True,
                  use_vocab=True,
                  tokenize=str.split,
                  lower=True,
                  batch_first=True,
                  fix_length=20)

LABEL = data.Field(sequential=False,
                   use_vocab=False,
                   batch_first=False,
                   is_target=True)
```

sequential : 시퀀스 데이터 여부 (True 기본값)

use_vocab : 단어 집합 만들 것인지 여부 (True 기본값)

tokenize : 토큰화 함수 지정 (string.split 기본값)

lower : 영어 데이터 전부 소문자화 (False 기본값)

batch_first : 미니 배치 차원 맨 앞 (False 기본값)

is_target : 레이블 데이터 여부 (False 기본값)

fix_length : 패딩 최대 허용 길이

TORCHTEXT

◆ DataSet 생성 - torchtext.legacy.data.TabularDataset

❖ 데이터 로딩 및 토큰화 수행 후 Dataset 생성 ➔ torchtext v0.10.0, 0.11.0

```
train_data, test_data = TabularDataset.splits(
        path='.', train='train_data.csv', test='test_data.csv', format='csv',
        fields=[('text', TEXT), ('label', LABEL)], skip_header=True)
```

- 정형 데이터파일로부터 직접 데이터를 읽을 때 유용
  - path : 데이터 파일 위치 경로 설정
  - format : 데이터 파일 포맷
  - fields : 정의한 필드 지정, 필드이름 지정
  - **skip_header : 첫번째 줄 데이터 무시 여부 설정**

TORCHTEXT

◆ 단어 집합(Vocabulary) 생성

❖ Tokenizer 생성

torchtext.data.utils.**get_tokenizer(** tokenizer , language='en' **)**

➔ **반환 : tokenizer 인스턴스**

- tokenizer : 토큰화 진행 할 tokenizer 함수 이름
  - → None : split( )
  - → 'basic_English' : basic_english_normalize( )
  - → tokenizer library : 라이브러리 관련 함수 반환
- language : 토큰화 언어 (기: en)

TORCHTEXT

◆ 단어 집합(Vocabulary) 생성

❖ Voca 객체 생성 : 토큰을 인덱스와 매핑, 토큰 ⇔ 정수인덱스 변환

CLASS torchtext.vocab.**Vocab(vocab)**

- get_stoi() : token을 정수인덱스로 반환
- get_itos() : 정수 인덱스를 token으로 반환
- __getitem__() : token에 매핑되는 정수인덱스값 반환
- lookup_token() : 정수인덱스에 매핑되는 token 반환
- forward() : encode 진행(문장 → 토큰화 → id값 변경) , nn.Module의 forward()
- lookup_indices() : encode 진행(문장 → 토큰화 → id값 변경)
- lookup_tokens() : decode 진행(id → 적합한 토큰)

TORCHTEXT

◆ 단어 집합(Vocabulary) 생성

❖ Voca 객체 생성 : 토큰을 인덱스와 매핑

torchtext.vocab.**vocab(** ordered_dict, min_freq=1, specials=None, special_first=True **)**

➔ **반환 : Vocab 인스턴스**

- Ordered_dict : 토큰을 해당 발생 빈도에 매핑하는 순서가 지정된 사전
- min_freq : 어휘에 토큰을 포함하는 데 필요한 최소 빈도
- Specials : 추가할 특수 기호. 공급된 토큰의 순서는 유지됨
- Special_first : 기호를 처음에 삽입할지 아니면 끝에 삽입할지 결정

TORCHTEXT

◆ 단어 집합(Vocabulary) 생성

❖ Voca 객체 생성 : iterator이용하여 생성 ➔ v0.11.0

torchtext.legacy.data.Field.**build_vocab(** iterator, min_freq = 1, max_size =1000, vectors=None **)** ➔ **반환 Vocab**

- 임의로 특별 토큰인 <unk>와 <pad> 추가 ➔ <unk>번호는 0번, <pad>번호는 1번 부여
  - iterator : Vocab 빌드 반복자, 토큰 목록 또는 반복자 생성
  - min_freq : Vocab에 토큰을 포함하기 위한 최소 빈도수
  - max_size : 최대 vocab_size 지정 (미지정시 전체 단어사전 개수 대입)
  - vectors : 워드임베딩 vector 지정, None으로 지정시 vector 사용 안함

TORCHTEXT

◆ 단어 집합(Vocabulary) 생성

❖ Voca 객체 생성 : iterator이용하여 생성 ➔ v0.18.0

torchtext.vocab.**build_vocab_from_iterator(** iterator, min_freq = 1, specials = None, special_first = True, max_tokens = None **)** ➔ **반환 Vocab**

- iterator : Vocab 빌드 반복자, 토큰 목록또는 반복자 생성
- min_freq : Vocab에 토큰을 포함하기 위한 최소 빈도수
- specials : 추가할 특수 기호
- special_first : 추가 특수 기호 처음 또는 끝 삽입여부 결정
- max_tokens : max_tokens - len(specials)값으로 최대 토큰 수

TORCHTEXT

◆ 데이터로더 생성 ➔ v0.11.0

- 데이터셋에서 미니 배치만큼 데이터를 로드하게 만들어주는 역할
- Iterator를 사용하여 데이터로더 생성

```
from torchtext.legacy.data import Iterator

batch_size = 5

train_loader = Iterator( dataset=train_data,  batch_size = batch_size )
test_loader = Iterator( dataset=test_data,  batch_size = batch_size )
```

TORCHTEXT

◆ 데이터로더 생성 ➔ v0.18.0

- 데이터셋에서 미니 배치만큼 데이터를 로드하게 만들어주는 역할
- Iterator를 사용하여 데이터로더 생성

```
from torch.utils.data import DataLoader


# 데이터 로더 인스턴스 생성
dataloader = DataLoader( train_iter,
                         batch_size=8,
                         shuffle=False )
```

PART I

RECURRENT NEURAL NETWORK

RECURRENT NEURAL NETWORK

◆ 순환 신경망(RNN: Recurrent Neural Network)

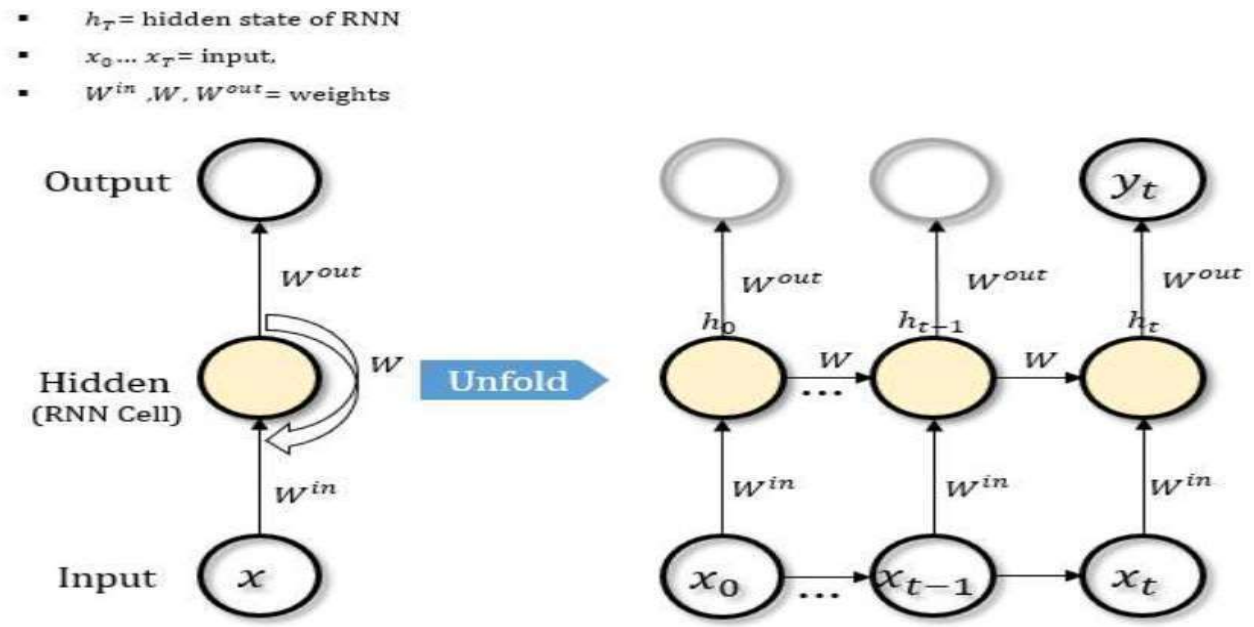
- **시간적으로 연속성이 있는 데이터**를 처리하기 위한 최적화된 인공신경망
- 입력과 출력을 **시퀀스 단위로 처리하는 시퀀스(Sequence) 모델**
- 현재까지 입력 데이터를 요약한 정보 즉, 기억 기능 가지는 신경망
- 새로운 입력 들어올 때마다 기억 조금씩 수정 ➔ 최종 남겨진 기억 : 모든 입력 전체 요약 정보

- 이전 상태에 대한 정보를 메모리 형태로 저장 가능
- 시계열 데이터, 순서가 중요한 데이터에 적합
- 인간의 **언어의 앞뒤 문맥**을 가지고 단어 예측하는 경우 적합함

RECURRENT NEURAL NETWORK

◆ 순환 신경망(RNN: Recurrent Neural Network)

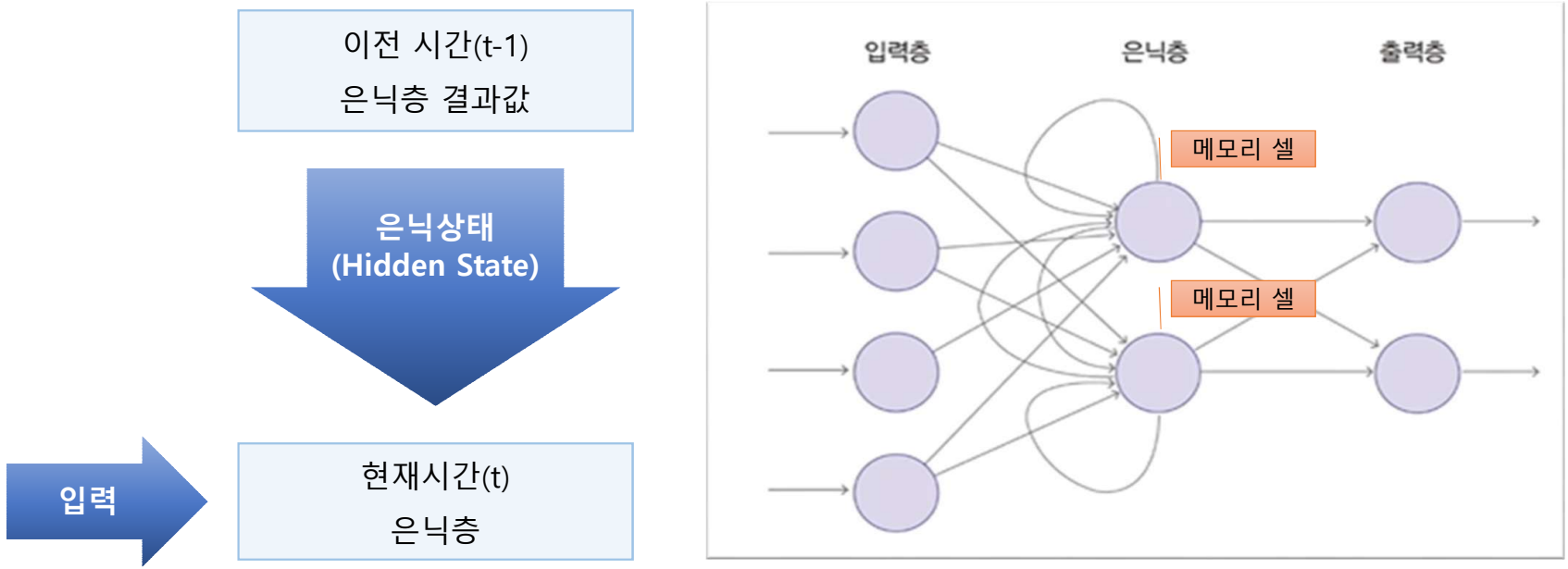
❖ 구조



RECURRENT NEURAL NETWORK

◆ 순환 신경망(RNN: Recurrent Neural Network)

❖ 구조



RECURRENT NEURAL NETWORK

◆ 순환 신경망(RNN: Recurrent Neural Network)

❖ 구성 요소

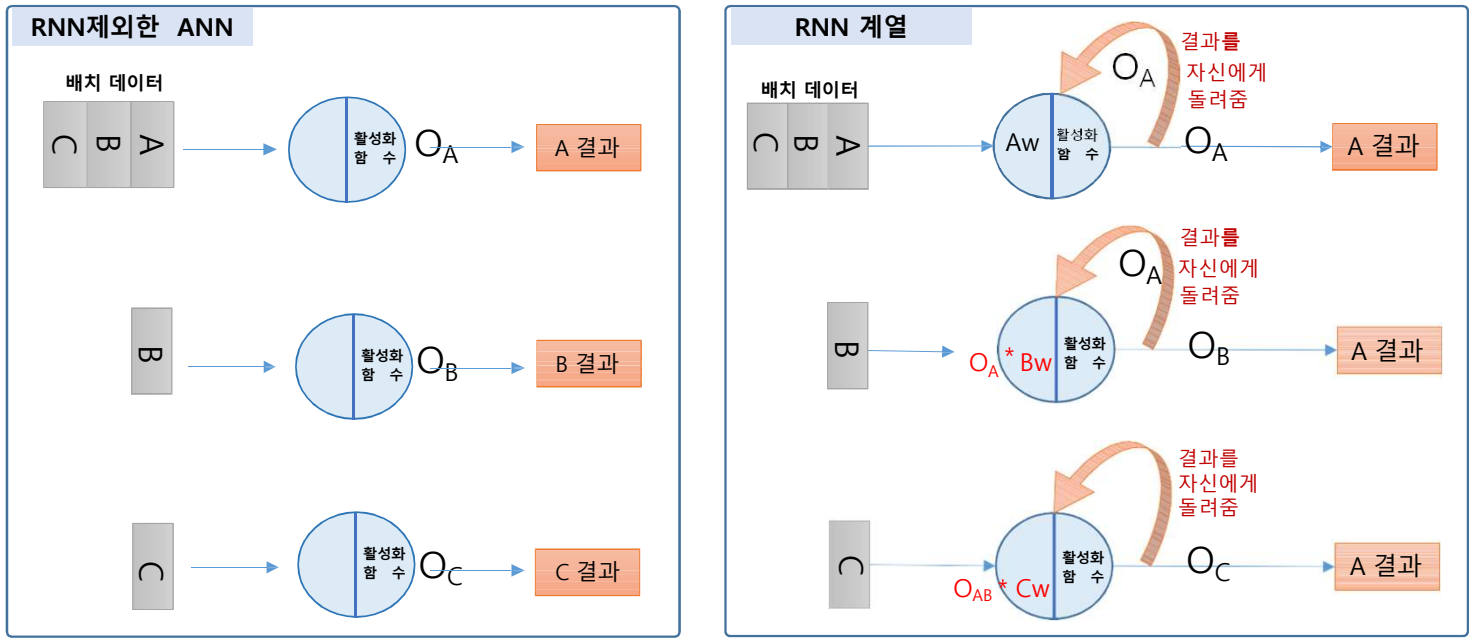
- 타입스텝(Timestep) : 1개 샘플을 처리하는 단위 (1문장 => 구성 단어 수)
- 쉘(Cell) : 이전 샘플 데이터 처리 결과 기억 요소

  RNN 은닉층에 존재
- 은닉상태(Hidden State) : 은닉층 출력 결과값 (CNN의 특성맵에 해당)

RECURRENT NEURAL NETWORK

◆ 순환 신경망(RNN: Recurrent Neural Network)

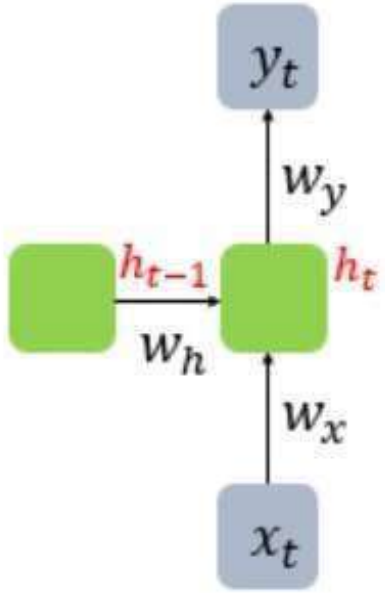
❖ 동작방식



RECURRENT NEURAL NETWORK

◆ 순환 신경망(RNN: Recurrent Neural Network)

❖ 매개변수



- 은닉층(Hidden Layer) : $ht=\tanh(Wxx_t + Whh_{t-1} + b)$
- 출력층(Out Layer) : $yt=F(Wyht + b)$

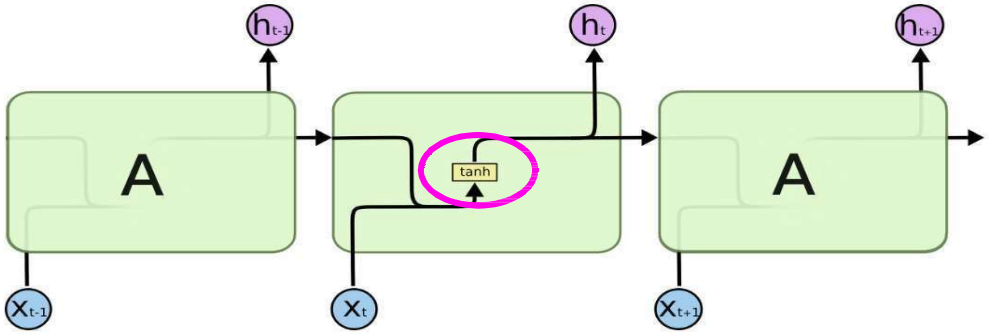
- 활성화 함수 : Tanh, ReLU
- h_{t-1} 초기값 : 0으로 초기화

- 입력데이터 : 벡터, 행렬 연산 진행
- 가중치 W_x, W_h, W_y : 같은 Layer에서는 동일 값

RECURRENT NEURAL NETWORK

◆ 순환 신경망(RNN: Recurrent Neural Network)

❖ 활성화함수

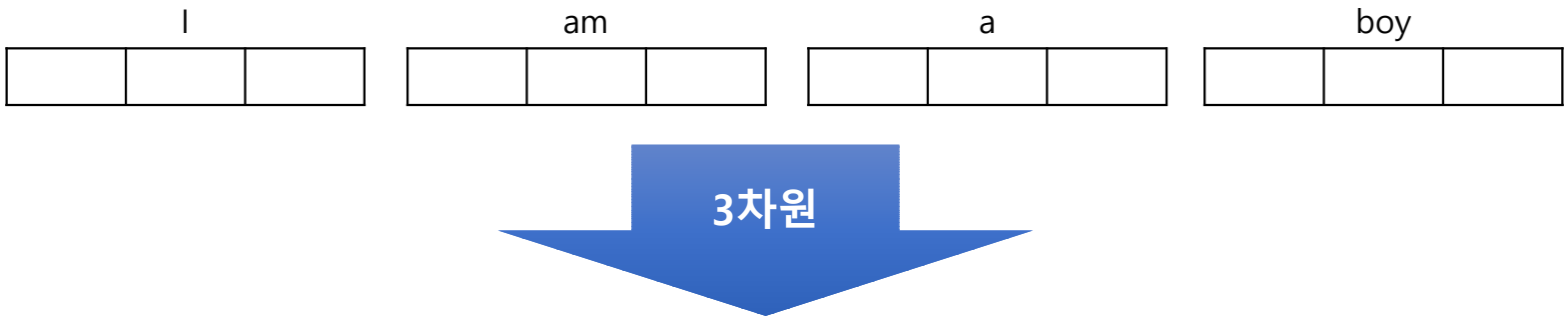


- 하이퍼볼릭탄젠트
- Sigmoid Function과 동일한 형태
- 출력값 : -1 부터 1까지
- 최대값이 1이기 때문에 Vanishing Gradient 현상 완화

RECURRENT NEURAL NETWORK

◆ 순환 신경망(RNN: Recurrent Neural Network)

❖ 입력 데이터



(Batch Size, Timestep Size, Token Size) ➔ (1, 4, 3)

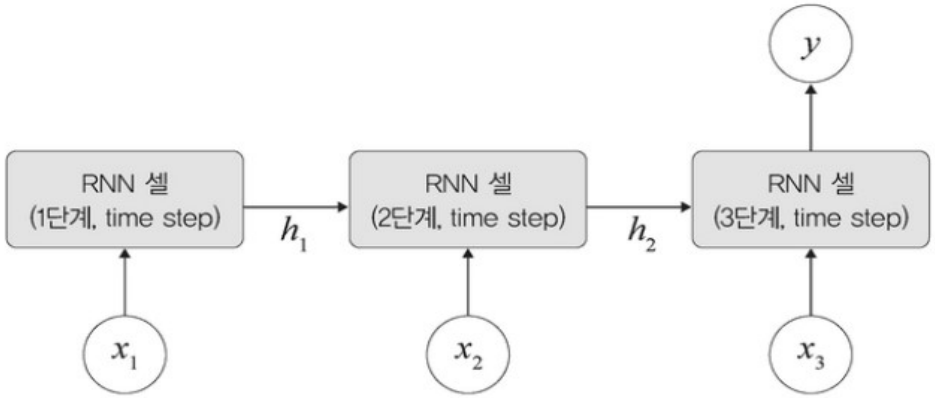
RECURRENT NEURAL NETWORK

◆ 순환 신경망(RNN: Recurrent Neural Network)

❖ 다양한 구성 : 입려과 출력 길이 다르게 설계

- 다대일 구성

→ 입력 : 여러 개
→ 출력 : 1개
→ 적용 : 감성분류
스팸메일분류



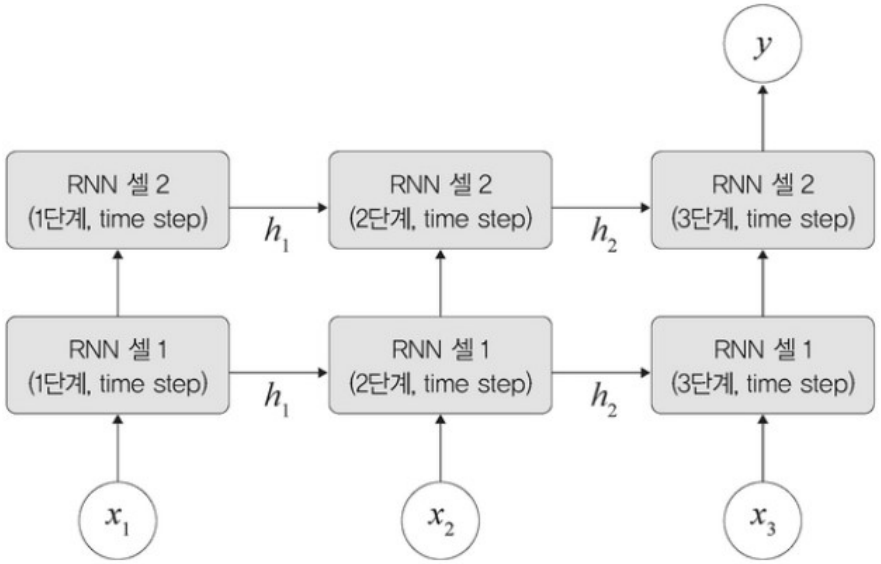
RECURRENT NEURAL NETWORK

◆ 순환 신경망(RNN: Recurrent Neural Network)

❖ 다양한 구성 : 입려과 출력 길이 다르게 설계

- 일대다 구성

→ 입력 : 1개
→ 출력 : 여러개
→ 적용 : 이미지 캡셔닝



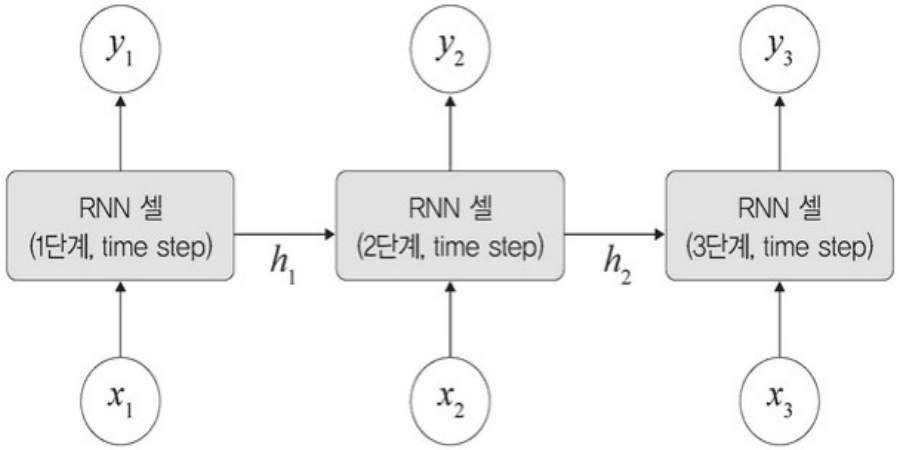
RECURRENT NEURAL NETWORK

◆ 순환 신경망(RNN: Recurrent Neural Network)

❖ 다양한 구성 : 입력과 출력 길이 다르게 설계

- 다대다 구성

→ 입력 : 여러개
→ 출력 : 여러개
→ 적용 : 챗봇
번역기
품사캐딩 & 개체명인식



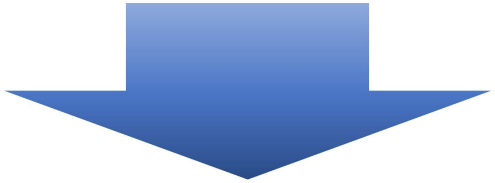
RECURRENT NEURAL NETWORK

◆ 순환 신경망(RNN: Recurrent Neural Network)

❖ 한계

먼 과거에 나온 정보에서 새로운 정보 유추 불가능

장기의존성문제 (Long Term Dependency)



LSTM (Long Short Term Memory)

- 장기의존성 문제 해결
- 역전파 시 기울기 소실 문제 해결

RECURRENT NEURAL NETWORK

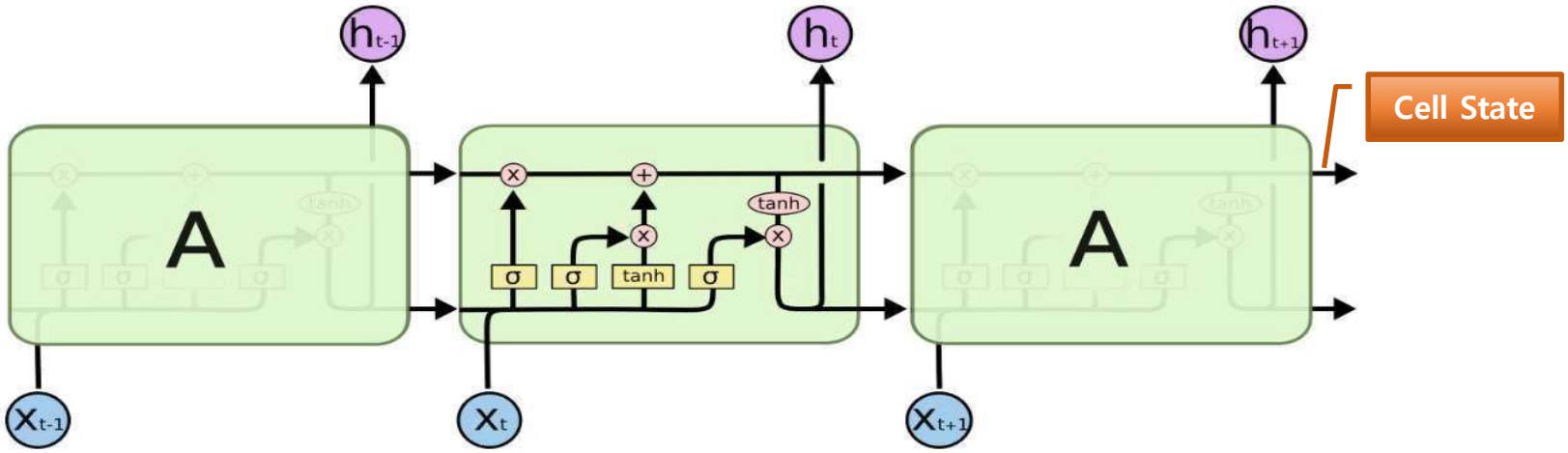
◆ LSTM : Long Short Term Memory

- RNN 장기의존성 문제, 기울기 소실 문제 해결한 알고리즘 중 하나
- RNN보다 복잡한 구조
- **과거의 기억을 보존하되, 필요가 없어진 기억을 지워버리는 기능** 추가
- RNN의 기울기 소실/폭주 문제 해결 요소 즉, Cell State 추가

RECURRENT NEURAL NETWORK

◆ LSTM : Long Short Term Memory

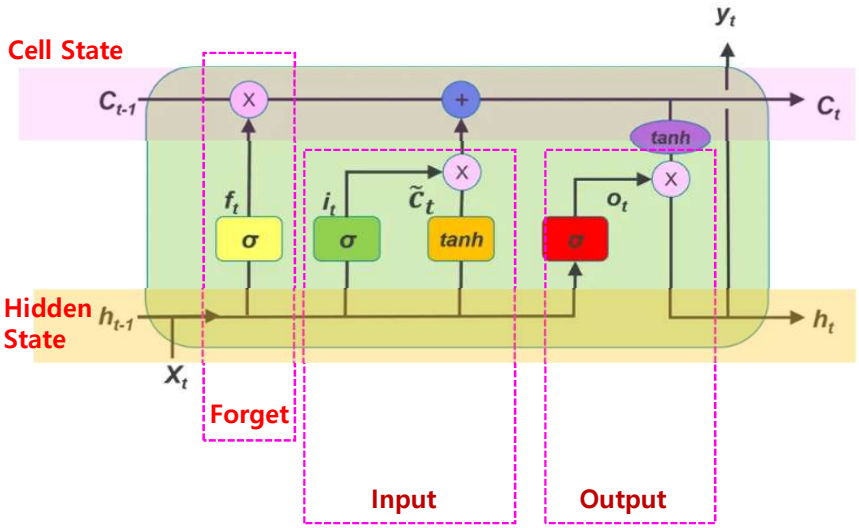
❖ 구조 : Hidden State + Cell State + 4 Gates



RECURRENT NEURAL NETWORK

◆ LSTM : Long Short Term Memory

❖ 구조 : Hidden State + Cell State + 4 Gates

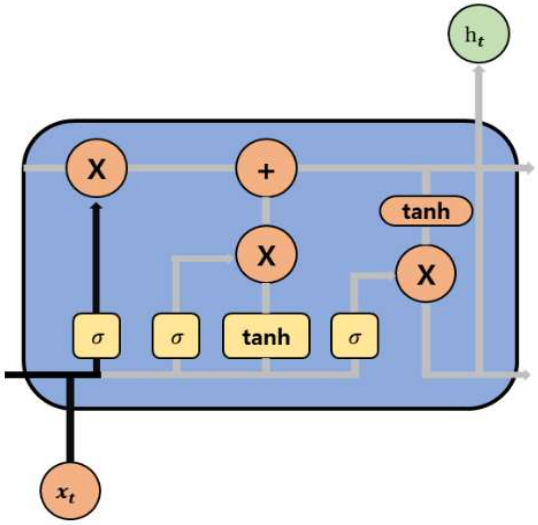


- **GATE**
  - → forget gate : 기존 정보 유지 여부 결정, 0이면 과거 기억 사라짐
  - → input gate : 현재 정보 기억 정도 결정 게이트
  - → output gate : 시그모이드 함수로 동작 제어
- **Cell State**
  - → forget과 input gate에 값을 **덧셈**하여 다음 Cell state 입력값 전달
  - → gate를 통해서 정보 필터링 기능

RECURRENT NEURAL NETWORK

◆ LSTM : Long Short Term Memory

❖ Forget Gate



▪ **기존 정보를 얼마나 잊어버릴지 결정하는 Gate**

→ sigmoid(hidden state) : 0 ~ 1 사이 값

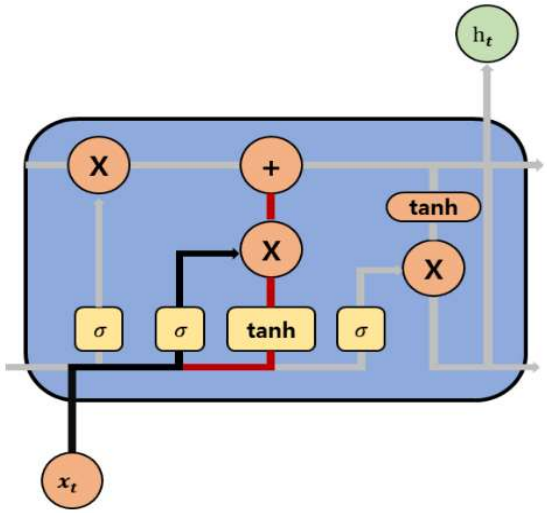
기존 정보 얼마나 사용할지 결정

→ 이전 Timestep의 Cell state와 곱셈 ➔ 새로운 Cell State 생성

RECURRENT NEURAL NETWORK

◆ LSTM : Long Short Term Memory

❖ Input Gate



▪ **새로운/현재 정보를 얼마나 기억할 지 결정하는 Gate**

→ sigmoid(새로운 정보) : 0 ~ 1 사이 값으로 새로운 정보 얼마나 사용할지 결정

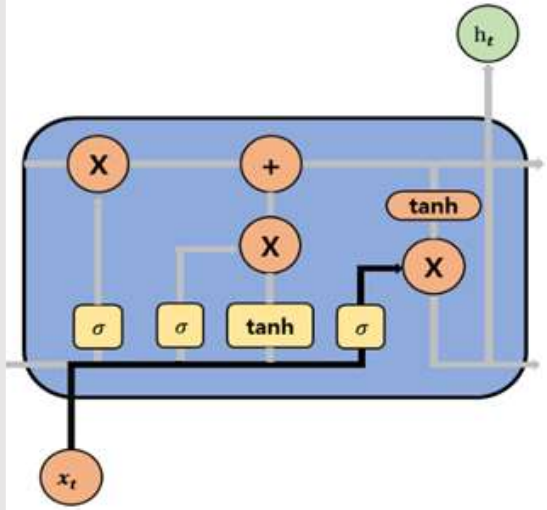
→ tanh(hidden state) : - 1 ~ 1 사이값으로 이전 상태값 얼마나 사용할지 결정

→ 새로운 정보와 Hidden state 곱셈 연산 진행

RECURRENT NEURAL NETWORK

◆ LSTM : Long Short Term Memory

❖ Output Gate



▪ **다음 층으로 전달할 hidden state를 만드는 Gate**

→ sigmoid(hidden state*wh) : 0 ~ 1 사이 값. 기존 정보 얼마나 사용할지 결정

→ sigmoid(입력값*wh) : 0 ~ 1 사이 값. 새로운 정보 얼마나 사용할지 결정

→ tanh(현재 Cell State) : 현재 Cell State 사용할 값 결정

→ 곱셈 연산 : 새로운 hidden state 생성

RECURRENT NEURAL NETWORK

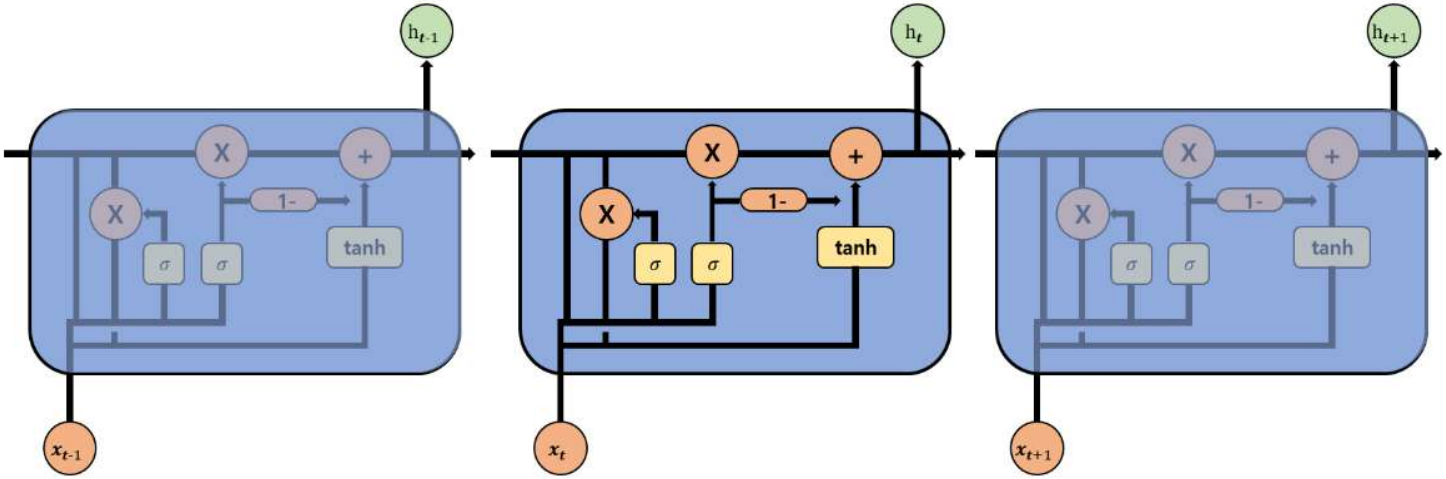
◆ GRU : Gated Recurrent Unit

- 2014년 뉴욕대학교 조경현 교수님이 집필한 논문에서 제안
- LSTM 장기 의존성 문제 해결책 유지하며, 은닉 상태 업데이트 계산 줄여줌
- GRU는 성능은 LSTM과 유사하면서 복잡했던 LSTM 구조 간단화
- LSTM의 Cell State를 없애고 Hidden State만 활용한 모델
- Reset Gate와 Update Gate만 존재

RECURRENT NEURAL NETWORK

◆ GRU : Gated Recurrent Unit

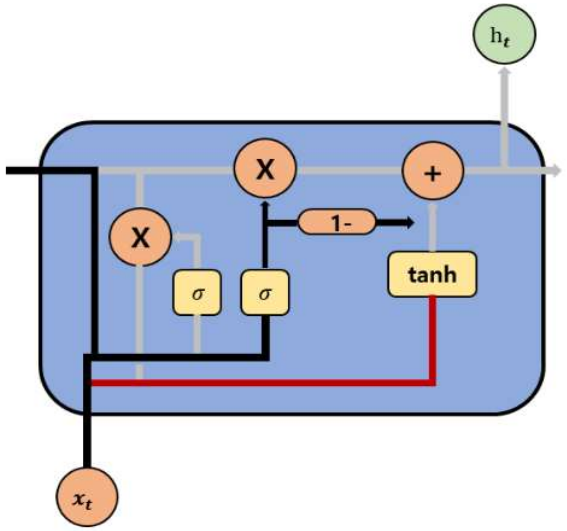
❖ 구조 : Update Gate (Forget Gate + Input Gate) + Reset Gate



RECURRENT NEURAL NETWORK

◆ GRU : Gated Recurrent Unit

❖ Update Gate



▪ **이전의 정보를 얼마나 통과시킬지 결정하는 Gate**

→ Forget Gate와 Input Gate를 합친 Gate

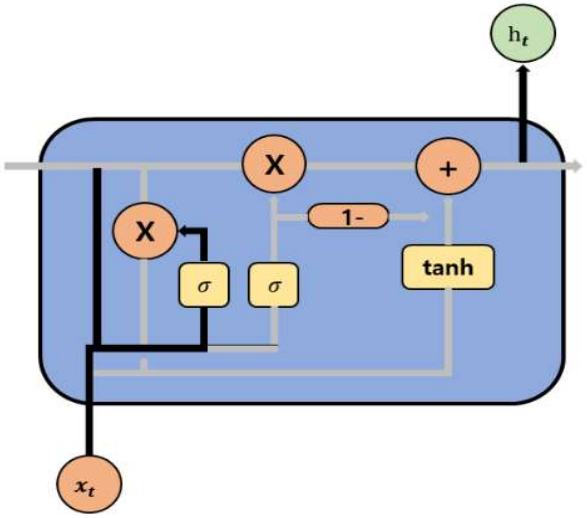
→ sigmoid(입력값*wh) : input gate 역할

→ sigmoid(hidden state*wh) : forget gate 역할

RECURRENT NEURAL NETWORK

◆ GRU : Gated Recurrent Unit

❖ Reset Gate



▪ **이전 hidden state 정보를 얼마나 잊을 지를 결정하는 Gate**

→ sigmoid() 함수 : 0 ~ 1 사이 범위의 잊을 정보양 결정

PART I

PYTORCH NLP LAYERS

PYTORCH NLP LAYERS

◆ Word Embedding

- 단어를 벡터로 표현하는 것으로 **밀집 표현(dense representation)**
- 벡터의 차원 => 사용자가 설정한 값으로 모든 단어의 벡터 표현의 차원 설정
- 0과 1만 가진 값이 아니라 실수값 가짐
- 밀집 벡터를 워드 임베딩 과정을 통해 나온 결과 => **Embedding Vector**
- 적용모델 : LSA, Word2Vec, FastText, Glove

- 파이토치
  ① 훈련 데이터로부터 처음부터 임베딩 벡터를 학습하는 방법
  ② 미리 사전에 훈련된 임베딩 벡터(pre-trained word embedding) 활용하는 방법

PYTORCH NLP LAYERS

◆ Word Embedding

❖ Embedding Layer

- 조건 : 입력 시퀀스의 각 단어들은 모두 정수 인코딩 되어 있어야 함
- 결과 : 입력 정수에 대해 밀집 벡터(dense vector)로 맵핑
- 학습 : 인공 신경망의 학습 과정에서 가중치가 학습되는 것과 같은 방식으로 훈련

```
import torch.nn as nn

# num_embeddings  : 임베딩 할 단어들 개수, 단어 집합 크기
# embedding_dim   : 임베딩 할 벡터 차원, 사용자 지정
# padding_idx     : 패딩 위한 토큰의 인덱스
embedding_layer = nn.Embedding(num_embeddings=len(vocab),
                               embedding_dim=3,
                               padding_idx=1)
```

PYTORCH NLP LAYERS

◆ Recurrent Layers

❖ RNN Layer & RNN CELL

`nn.RNNBase`	Base class for RNN modules (RNN, LSTM, GRU).
`nn.RNN`	Apply a multi-layer Elman RNN with $\tanh$ or ReLU non-linearity to an input sequence.
`nn.LSTM`	Apply a multi-layer long short-term memory (LSTM) RNN to an input sequence.
`nn.GRU`	Apply a multi-layer gated recurrent unit (GRU) RNN to an input sequence.
`nn.RNNCell`	An Elman RNN cell with tanh or ReLU non-linearity.
`nn.LSTMCell`	A long short-term memory (LSTM) cell.

PYTORCH NLP LAYERS

◆ Recurrent Layers

❖ RNN Class

```
CLASS  torch.nn.RNN(input_size, hidden_size, num_layers=1,
       nonlinearity='tanh', bias=True, batch_first=False, dropout=0.0,
       bidirectional=False, device=None, dtype=None)  [SOURCE]
```

→ input_size : 특성 값의 개수

→ hidden_size : hidden state 개수

→ num_layers: RNN 레이어 층을 쌓을 경우 지정 ➔ stacking RNN

→ batch_first : 입력 데이터 shape에서 첫번째 차원을 batch로 설정 ➔ False

→ dropout : 마지막 layer 제외한 각 RNN 레이어의 출력에 dropout 설정

→ bidirectional : 양방향 RNN 설정

PYTORCH NLP LAYERS

◆ Recurrent Layers

❖ LSTM Class

CLASS torch.nn.LSTM(*input_size, hidden_size, num_layers=1, bias=True, batch_first=False, dropout=0.0, bidirectional=False, proj_size=0, device=None, dtype=None*) [SOURCE]

❖ GRU

CLASS torch.nn.GRU(*input_size, hidden_size, num_layers=1, bias=True, batch_first=False, dropout=0.0, bidirectional=False, device=None, dtype=None*) [SOURCE]

PART I

VARIOUS NLP PRACTICES

VARIOUS NLP PRACTICES

◆ 감성분석 : Sentimetal Analysis

- 텍스트에 들어있는 의견 감성, 평가, 태도 등 주관적인 정보 분석 과정
- 가정

  → 비슷한 감정 표현하는 문서는 유사한 단어 구성 및 언어적 특징 보일 것이다.

- 분석방법

  → 텍스트 내 감정 분류

  → 긍정/부정의 정도 점수화

VARIOUS NLP PRACTICES

◆ 감성분석 : Sentimetal Analysis

❖ IMDB 데이터셋

- 인터넷영화리뷰 사이트
- 25,000개 리뷰
- 긍정/부정 평가

VARIOUS NLP PRACTICES

◆ 감성분석 : Sentimetal Analysis

❖ IMDB 데이터셋 전처리 과정

VARIOUS NLP PRACTICES

◆ 감성분석 : Sentimetal Analysis

❖ IMDB 데이터셋 전처리 과정

VARIOUS NLP PRACTICES

◆ 감성분석 : Sentimetal Analysis

❖ 긍정/부정 리뷰 평가 모델