

한양대 분석가 교육과정

Introduction to NLP With PyTorch

2021년 06월 16일

비알프레임

*Machine Learning &
Predictive Analytics*



03. Introduction to NLP with PyTorch

- **학습 목표**

- 자연어 처리 작업(Natural Language Processing)을 위한 텍스트 처리 방법의 이해
- 순환 신경망(Recurrent Neural Network)과 생성 신경망(Generative Neural Networks)의 소개
- 어텐션 메커니즘(Attention Mechanisms) 학습
- 텍스트 분류 모델 구축 방법 학습

- **전제 조건**

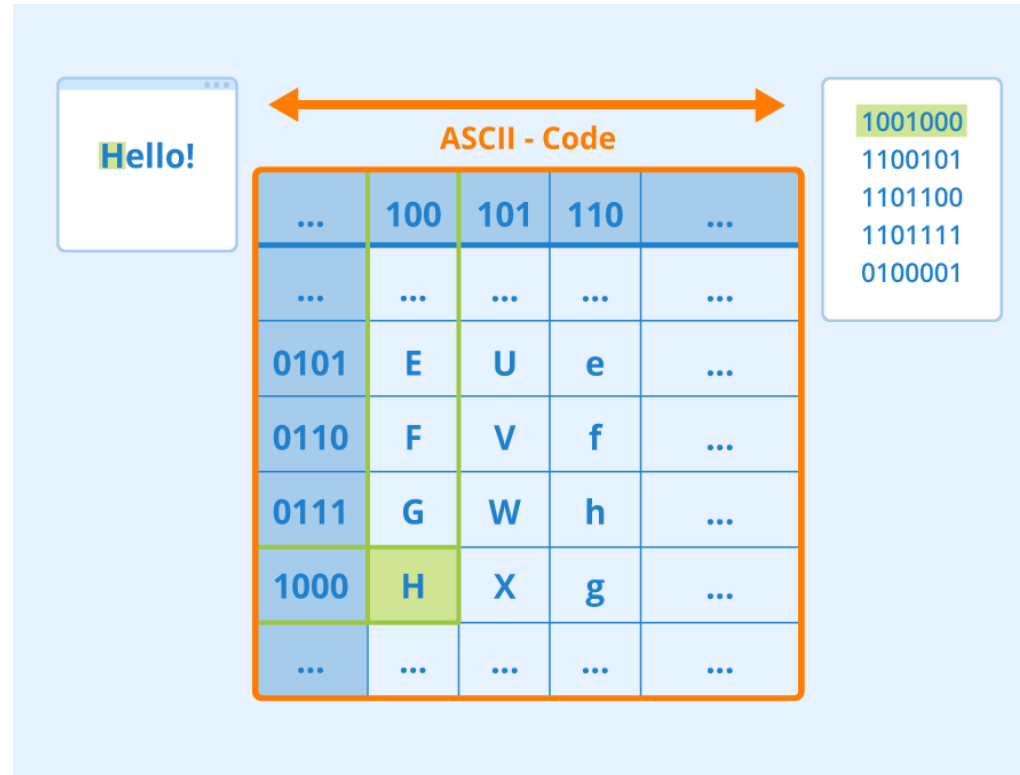
- 기본 Python 지식
- Jupyter Notebooks 사용 방법에 대한 기본 지식
- 기본 머신러닝의 이해

03. Introduction to NLP with PyTorch

• 1. 텐서로서 텍스트 표현

• 텍스트 표현(Representing Text) 배경

- 신경망으로 자연어 처리(NLP) 작업을 해결하려면 텍스트를 텐서로 나타낼 수 있는 방법이 필요하다.
- 컴퓨터는 이미 ASCII 또는 UTF-8과 같은 인코딩을 사용하여 화면의 글꼴에 매핑되는 숫자로 텍스트 문자를 나타냅니다.
- 우리는 각 글자가 무엇을 나타내는지, 그리고 어떻게 모든 글자가 함께 모여 문장의 단어를 형성하는지 이해하지만 모델은 그러한 이해를 가지고 있지 않으며, 신경망은 훈련 중에 그 의미를 배워야 한다.



03. Introduction to NLP with PyTorch

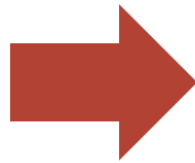
- 1. 텐서로서 텍스트 표현

- 텍스트 표현(Representing Text) 접근방식

- 문자 수준 표현(Character-level representation) : 각 문자를 숫자로서 처리하여 텍스트를 나타낼 때 사용하는 방식으로 문자별 C 길이의 텐서가 주어졌다면, "HELLO"는 $5 \times C$ 텐서로 표현할 수 있다.

- 단어 수준 표현(Word-level representation) : 특정 텍스트에서 모든 단어의 사전(Vocabulary)를 생성하여 원-핫 인코딩으로 단어를 표현하는 방법, 이 접근 방식은 각 문자 자체에는 큰 의미가 없기 때문에 문자 수준 표현보다 더 나은 편이며, 따라서 상위 수준의 의미 개념인 단어를 사용함으로써 신경망에 대한 작업을 단순화한다.

Vocabulary:
Man, woman, boy,
girl, prince,
princess, queen,
king, monarch



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|----------|---|---|---|---|---|---|---|---|---|
| man | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| woman | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| boy | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| girl | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| prince | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| princess | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| queen | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| king | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| monarch | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Each word gets
a 1x9 vector
representation

03. Introduction to NLP with PyTorch

• 2. 텍스트 분류(Text Classification) 작업

- torchtext 모듈을 임포트하여 AG_NEWS 데이터셋 로드

※ AG_NEWS dataset : 뉴스 헤드라인 토픽을 [World, Sports, Business, Sci/Tech] 와 같이 4가지 클래스로 구성된 데이터셋
각 클래스 별 학습 데이터셋 : 30,000개 / 테스트 : 1,900개의 텍스트로 구성

```
import torch
import torchtext
import os
import collections
os.makedirs('./newsdata', exist_ok=True)
train_dataset, test_dataset = torchtext.datasets.AG_NEWS(root='./data')
classes = ['World', 'Sports', 'Business', 'Sci/Tech']
```

```
for i, x in zip(range(5), train_dataset):
    print(f"**{classes[x[0]]}** -> {x[1]}")
```

Sci/Tech -> Carlyle Looks Toward Commercial Aerospace (Reuters) Reuters - Private investment firm Carlyle Group, which has a reputation for making well-timed and occasionally controversial plays in the defense industry, has quietly placed its bets on another part of the market.

Sci/Tech -> Oil and Economy Cloud Stocks' Outlook (Reuters) Reuters - Soaring crude prices plus worries about the economy and the outlook for earnings are expected to hang over the stock market next week during the depth of the summer doldrums.

03. Introduction to NLP with PyTorch

• 2. 텍스트 분류(Text Classification) 작업

- 불러온 텍스트 데이터셋을 텐서로 표현하기 위해 숫자로 변환

1) 텍스트를 토큰(token)으로 분리하기 위하여 tokenizer 사용

- torch.text.utils.get_tokenizer 메서드를 사용하여 텍스트 토큰으로 분리

```
tokenizer = torchtext.data.utils.get_tokenizer('basic_english')
tokenizer('He said: hello')
```

2) 토큰의 사전(vocabulary) 구축

- collections.Counter : 컨테이너에 동일한 값의 자료가 몇 개인지를 파악하는데 사용하는 객체
- torchtext.vocab.Vocab을 통하여 vocabulary 구축

```
counter = collections.Counter()
for (label, line) in train_dataset:
    counter.update(tokenizer(line))
vocab = torchtext.vocab.Vocab(counter, min_freq=1)
```

- encode 라는 함수를 정의하여 텍스트 인코딩 결과 확인

```
encode('I love to play with my words') ➡ [283, 2321, 5, 337, 19, 1301, 2357]
```

03. Introduction to NLP with PyTorch

• 2. 텍스트 분류(Text Classification) 작업

- Bag of Words(BoW)를 이용한 텍스트 표현

- Bag of Words : 단어들의 순서는 전혀 고려하지 않고, 단어들의 출현 빈도(frequency)에만 집중하는 텍스트 데이터의 수치화 표현 방법이다. 단어는 의미를 나타내기 때문에, 우리는 때때로 문장의 순서와 상관없이 각각의 단어들을 보는 것만으로도 텍스트의 의미를 알아낼 수 있다.

※ 뉴스 토픽 분류 예시 : snow → 일기 예보 / stocks, dollars → 금융 뉴스

- BoW 예시

- 각 단어에 대한 원-핫 인코딩 벡터의 총합과 같다.



- "banana, banana, mango" → [0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 1, 0, 0,]

03. Introduction to NLP with PyTorch

- 2. 텍스트 분류(Text Classification) 작업

- Bag of Words(BoW)를 이용한 텍스트 표현
 - Sickit-learn 패키지에서 제공하는 CountVectorizer를 사용하여 단어 표현

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
corpus = [
    'I like hot dogs.',
    'The dog ran fast.',
    'Its hot outside.',
]
vectorizer.fit_transform(corpus)
vectorizer.transform(['My dog likes hot dogs on a hot day.']).toarray()
```

- 최종 단어 표현을 위한 사용자 지정 함수(to_bow) 정의

```
vocab_size = len(vocab)

def to_bow(text, bow_vocab_size=vocab_size):
    res = torch.zeros(bow_vocab_size, dtype=torch.float32)
    for i in encode(text):
        if i < bow_vocab_size:
            res[i] += 1
    return res
```


03. Introduction to NLP with PyTorch

- 2. 텍스트 분류(Text Classification) 작업

- Bag of Words(BoW) Classifier 학습하기

- 모든 positional vector를 BoW 표현으로 변환하고 미니 배치 별로 묶어주기 위하여 bowify 함수 정의
- 앞서 정의한 bowify를 collate_fn으로 정의하여 학습 및 테스트 데이터셋에 대한 Dataloader 생성

```
from torch.utils.data import DataLoader
import numpy as np

# this collate function gets list of batch_size tuples, and needs to
# return a pair of label-feature tensors for the whole minibatch
def bowify(b):
    return (
        torch.LongTensor([t[0]-1 for t in b]),
        torch.stack([to_bow(t[1]) for t in b])
    )

train_loader = DataLoader(train_dataset, batch_size=16, collate_fn=bowify, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=16, collate_fn=bowify, shuffle=True)
```

03. Introduction to NLP with PyTorch

- 2. 텍스트 분류(Text Classification) 작업

- Bag of Words(BoW) Classifier 학습하기 : 간단한 분류기 신경망 정의
 - 입력 벡터 사이즈 : vocab_size
 - 출력 벡터 사이즈 : 4
 - 활성화 함수 : LogSoftmax

```
net = torch.nn.Sequential(  
    torch.nn.Linear(vocab_size, 4),  
    torch.nn.LogSoftmax(dim=1)  
)
```

03. Introduction to NLP with PyTorch

- 2. 텍스트 분류(Text Classification) 작업

- Bag of Words(BoW) Classifier 학습하기 : 훈련 루프 정의
 - 데이터셋이 크기 때문에 학습은 1 epoch 만 시행

```
def train_epoch(net, dataloader, lr=0.01, optimizer=None, loss_fn = torch.nn.NLLLoss(), epoch_size=None, report_freq=200):
    optimizer = optimizer or torch.optim.Adam(net.parameters(), lr=lr)
    net.train()
    total_loss, acc, count, i = 0, 0, 0, 0
    for labels, features in dataloader:
        optimizer.zero_grad()
        out = net(features)
        loss = loss_fn(out, labels) #cross_entropy(out, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss
        _, predicted = torch.max(out, 1)
        acc += (predicted == labels).sum()
        count += len(labels)
        i += 1
        if i % report_freq == 0:
            print(f"{count}: acc={acc.item()/count}")
        if epoch_size and count > epoch_size:
            break
    return total_loss.item()/count, acc.item()/count
```

03. Introduction to NLP with PyTorch

- 2. 텍스트 분류(Text Classification) 작업

- N-gram representation

- BoW 표현의 한계점으로 특정 단어들은 다중 단어로 표현된다는 점이 있다

- 예를 들면, 서로 다른 의미를 가진 "hot"과 "dog"가 합쳐져 생성된 "hot dog" 대표적인 예

- 이러한 문제를 해결하기 위한 방법이 N-gram representation : 특정한 n개의 연속적인 단어를 사전(Vocabulary)에 추가

- 예시 문장 - An adorable little boy is spreading smiles

- 1) unigrams : an, adorable, little, boy, is, spreading, smiles

- 2) bigrams : an adorable, adorable little, little boy, boy is, is spreading, spreading smiles

- 3) trigrams : an adorable little, adorable little boy, little boy is, boy is spreading, is spreading smiles

- 4) 4-grams : an adorable little boy, adorable little boy is, little boy is spreading, boy is spreading smiles

03. Introduction to NLP with PyTorch

- 2. 텍스트 분류(Text Classification) 작업

- N-gram representation

- Sickit learn의 CountVectorizer 메서드에서 ngram_range = (1,2) 로 설정하여 bigram 표현

```
bigram_vectorizer = CountVectorizer(ngram_range=(1, 2), token_pattern=r'\b\w+\b', min_df=1)
corpus = [
    'I like hot dogs.',
    'The dog ran fast.',
    'Its hot outside.',
]
bigram_vectorizer.fit_transform(corpus)
print("Vocabulary:\n", bigram_vectorizer.vocabulary_)
bigram_vectorizer.transform(['My dog likes hot dogs on a hot day.']).toarray()
```

Vocabulary:

```
{'i': 7, 'like': 11, 'hot': 4, 'dogs': 2, 'i like': 8, 'like hot': 12, 'hot dogs': 5, 'the': 16,
'dog': 0, 'ran': 14, 'fast': 3, 'the dog': 17, 'dog ran': 1, 'ran fast': 15, 'its': 9, 'outside': 13,
'its hot': 10, 'hot outside': 6}
```

03. Introduction to NLP with PyTorch

- 2. 텍스트 분류(Text Classification) 작업

- N-gram representation
 - N-gram 접근법의 주요 단점은 사전(vocabulary)의 크기가 급격하게 커지는 점이다 → 단어 임베딩의 필요성
 - torchtext.data.utils.ngrams_iterator 을 이용하여 bigram 표현 생성

```
counter = collections.Counter()
for (label, line) in train_dataset:
    l = tokenizer(line)
    counter.update(torchtext.data.utils.ngrams_iterator(l, ngrams=2))

bi_vocab = torchtext.vocab.Vocab(counter, min_freq=1)

print("Bigram vocabulary length = ", len(bi_vocab))
```

```
Bigram vocabulary length = 1308844
```

03. Introduction to NLP with PyTorch

• 2. 텍스트 분류(Text Classification) 작업

- Term Frequency Inverse Document Frequency, TF-IDF

- 이전에서 다룬 BoW 표현에서, 단어 발생은 단어 종류에 관계없이 균등한 가중치로 집계된다. 그러나, a, in 등과 같이 문장에서 빈번하게 등장하는 단어들은 다른 전문 용어들보다 분류 문제에 훨씬 덜 중요하다는 것은 분명하다.

- TF-IDF는 단어의 빈도와 역 문서 빈도(문서의 빈도에 특정 식을 취함)를 사용하여 DTM 내의 각 단어들마다 중요한 정도를 가중치로 주는 방법이다.

- sickit-learn의 TfidfVectorizer를 사용하여 쉽게 생성 가능하다.

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(ngram_range=(1,2))
vectorizer.fit_transform(corpus)
vectorizer.transform(['My dog likes hot dogs on a hot day.']).toarray()
```

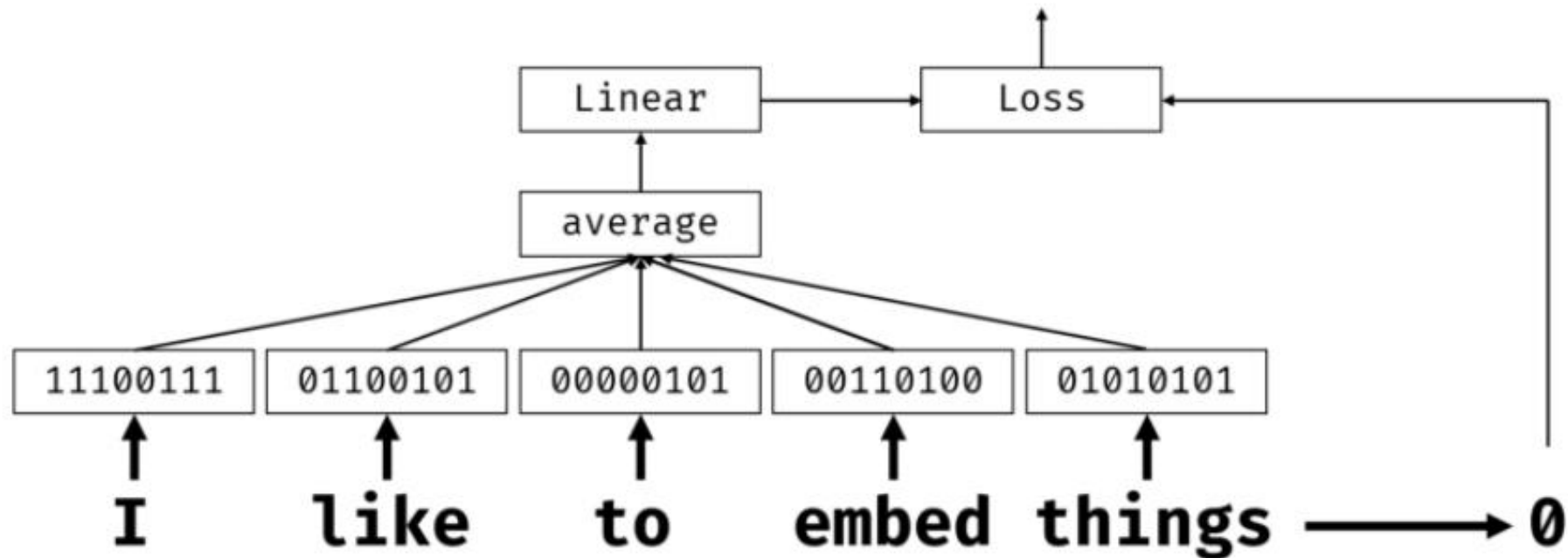
```
array([[0.43381609, 0.          , 0.43381609, 0.          , 0.65985664,
        0.43381609, 0.          , 0.          , 0.          , 0.          ,
        0.          , 0.          , 0.          , 0.          , 0.          ,
        0.          ]])
```

03. Introduction to NLP with PyTorch

• 3. 임베딩을 이용한 단어 표현

• 임베딩(Embeddings)

- 이전 섹션에서 사용한 원-핫 벡터(BoW 벡터) 표현은 메모리 측면에서 효율적이지 않으며, 단어 간의 의미적 유사성을 표현하지 못한다는 단점이 있다.
- 따라서 임베딩의 개념은 단어의 의미를 반영하는 저차원 밀도 벡터(lower-dimensional dense vector)로 단어 표현
- 임베딩 계층은 단어를 입력으로 받아 특정한 임베딩 사이즈를 지정하여 출력 벡터를 생성하는 작업을 수행하며, 모델의 첫번째 레이어에 사용하여 각 단어를 변환한 다음 합계 또는 평균 등 집계 함수를 사용 가능



03. Introduction to NLP with PyTorch

- 3. 임베딩을 이용한 단어 표현

- 임베딩 계층을 이용하여 간단한 분류기 모델 구축
 - torch.nn.Embedding 메서드를 이용하여 임베딩 계층 생성
 - 각 단어의 임베딩 벡터를 평균값을 취하여 최종 분류기에 통과

```
class EmbedClassifier(torch.nn.Module):  
    def __init__(self, vocab_size, embed_dim, num_class):  
        super().__init__()  
        self.embedding = torch.nn.Embedding(vocab_size, embed_dim)  
        self.fc = torch.nn.Linear(embed_dim, num_class)  
  
    def forward(self, x):  
        x = self.embedding(x)  
        x = torch.mean(x, dim=1)  
        return self.fc(x)
```

03. Introduction to NLP with PyTorch

• 3. 임베딩을 이용한 단어 표현

• 다양한 길이의 단어 시퀀스 다루기

- 이전 섹션에서 BoW 표현은 텍스트 시퀀스의 실제 길이의 상관없이 동일한 사이즈의 벡터로 표현했지만, 단어 임베딩에서는 텍스트 샘플에서 다양한 길이의 단어를 얻게되므로 이러한 샘플을 미니배치로 결합할 때 길이가 짧은 단어에 대해서는 패딩을 적용해야 한다.

- torch.nn.functional.pad 메서드를 이용하여 미니배치에서 가장 긴 단어의 길이만큼 나머지 단어의 토큰을 패딩
- 정의한 padify 함수를 collate_fn으로 정의하여 학습 데이터셋에 대한 DataLoader 생성

```
def padify(b):  
    # b is the list of tuples of length batch_size  
    # - first element of a tuple = label,  
    # - second = feature (text sequence)  
    # build vectorized sequence  
    v = [encode(x[1]) for x in b]  
    # first, compute max length of a sequence in this minibatch  
    l = max(map(len,v))  
    return ( # tuple of two tensors - labels and features  
            torch.LongTensor([t[0]-1 for t in b]),  
            torch.stack([torch.nn.functional.pad(torch.tensor(t),(0,l-len(t)),mode='constant',value=0) for t in v])  
            )
```

```
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=16, collate_fn=padify, shuffle=True)
```

03. Introduction to NLP with PyTorch

- 3. 임베딩을 이용한 단어 표현

- 임베딩 분류기 학습하기

- 정의한 모델 클래스(EmbedClassifier)를 불러와 학습 진행

- AG_news 데이터셋의 사전 길이와 임베딩 벡터 길이(32)를 설정하여 모델 구축

※ epoch_size(레코드 수)를 늘려서 전체 데이터셋에 대해 학습을 진행하면 분류 정확도가 더욱 올라간다.

```
net = EmbedClassifier(vocab_size, 32, len(classes)).to(device)
train_epoch(net, train_loader, lr=1, epoch_size=25000)
```

3200: acc=0.6428125

6400: acc=0.68453125

9600: acc=0.7123958333333333

12800: acc=0.725703125

16000: acc=0.7365625

19200: acc=0.7464583333333333

22400: acc=0.7548214285714285

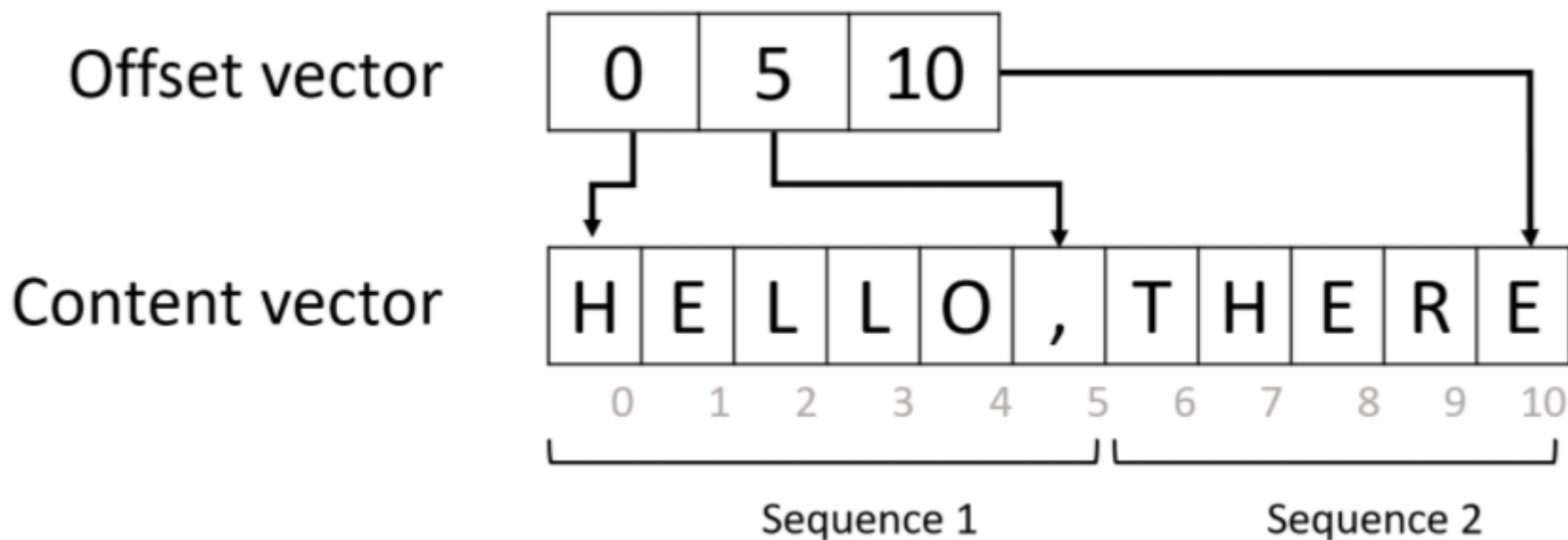
(0.9526769402541186, 0.7595969289827256)

03. Introduction to NLP with PyTorch

- 3. 임베딩을 이용한 단어 표현

- 임베딩 계층과 다양한 길이의 단어 시퀀스 표현

- 이전 접근방법에서 모든 시퀀스를 미니배치에 맞추기 위하여 동일한 길이로 패딩하는 방법은 가장 효율적인 방법이 아니다. 또 다른 접근방법은 하나의 큰 벡터에 저장된 모든 시퀀스의 오프셋 벡터를 사용하는 것이다.



03. Introduction to NLP with PyTorch

- 3. 임베딩을 이용한 단어 표현

- 임베딩 계층과 다양한 길이의 단어 시퀀스 표현

- torch.nn.EmbeddingBag 메서드를 사용하여 offset 표현을 사용할 수 있다. EmbeddingBag 계층은 기존 Embedding 계층과 유사하며 content 벡터와 offset 벡터를 입력으로 받는다.

```
class EmbedClassifier(torch.nn.Module):  
    def __init__(self, vocab_size, embed_dim, num_class):  
        super().__init__()  
        self.embedding = torch.nn.EmbeddingBag(vocab_size, embed_dim)  
        self.fc = torch.nn.Linear(embed_dim, num_class)  
  
    def forward(self, text, off):  
        x = self.embedding(text, off)  
        return self.fc(x)
```

03. Introduction to NLP with PyTorch

- 3. 임베딩을 이용한 단어 표현

- 임베딩 계층과 다양한 길이의 단어 시퀀스 표현
 - 모델 학습을 위한 데이터셋을 준비하기 위하여 offset 벡터를 만드는 offsetify 함수 정의
 - 단어 토큰의 모든 길이를 계산하고 누적합을 계산하여 offset 벡터 생성
 - 정의한 offsetify 함수를 collate_fn으로 정의하여 학습 데이터셋에 대한 DataLoader 생성

```
def offsetify(b):  
    # first, compute data tensor from all sequences  
    x = [torch.tensor(encode(t[1])) for t in b]  
    # now, compute the offsets by accumulating the tensor of sequence lengths  
    o = [0] + [len(t) for t in x]  
    o = torch.tensor(o[:-1]).cumsum(dim=0)  
    return (  
        torch.LongTensor([t[0]-1 for t in b]), # labels  
        torch.cat(x), # text  
        o  
    )  
  
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=16, collate_fn=offsetify, shuffle=True)
```

03. Introduction to NLP with PyTorch

• 3. 임베딩을 이용한 단어 표현

• 임베딩 분류기 학습하기

- 모델의 데이터로더가 label, embedding, offset 벡터를 불러오므로 모델 학습 함수도 약간의 수정을 거친다.

→ `out = net(text, off)`

```
net = EmbedClassifier(vocab_size, 32, len(classes)).to(device)

def train_epoch_emb(net, dataloader, lr=0.01, optimizer=None, loss_fn = torch.nn.CrossEntropyLoss(), epoch_size=None, report_freq=200):
    optimizer = optimizer or torch.optim.Adam(net.parameters(), lr=lr)
    loss_fn = loss_fn.to(device)
    net.train()
    total_loss, acc, count, i = 0, 0, 0, 0
    for labels, text, off in dataloader:
        optimizer.zero_grad()
        labels, text, off = labels.to(device), text.to(device), off.to(device)
        out = net(text, off)
        loss = loss_fn(out, labels) #cross_entropy(out, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss
        _, predicted = torch.max(out, 1)
        acc += (predicted == labels).sum()
        count += len(labels)
        i += 1
    if i % report_freq == 0:
        print(f"{count}: acc={acc.item()/count}")
    if epoch_size and count > epoch_size:
        break
    return total_loss.item()/count, acc.item()/count

train_epoch_emb(net, train_loader, lr=4, epoch_size=25000)
```

03. Introduction to NLP with PyTorch

3. 임베딩을 이용한 단어 표현

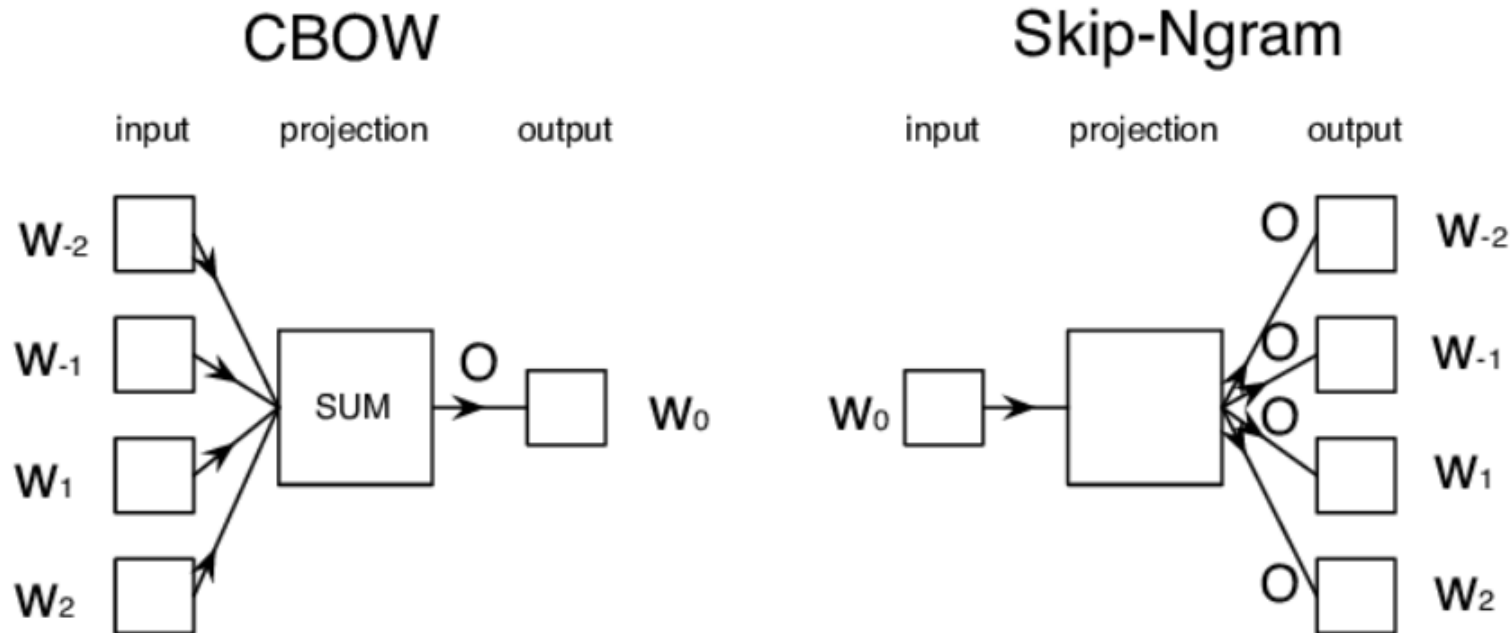
- Semantic Embeddings : Word2Vec

- 앞장에서 다룬 임베딩 방법은 서로 다른 길이의 시퀀스를 해결하기 위한 접근방법이고, 단어들의 다양한 의미론적 뜻(semantic meaning)을 갖지 못한다. 이를 위하여 대규모 텍스트 데이터셋에 대한 임베딩 모델을 사전 훈련할 필요가 있다.

- Semantic Embeddings을 훈련시키는 방법 중 하나는 Word2Vec이며, 단어의 분산 표현을 생성하는 2가지 주요 아키텍처는 다음과 같다.

- 1) Continuous bag-of-words(CBoW) : 문장 내에서 특정 단어의 주변 단어들로 해당 단어를 예측하는 방법

- 2) Continuous skip-gram : CBoW와는 반대로 특정 단어를 입력으로 주변 단어들을 예측하는 방법



03. Introduction to NLP with PyTorch

- 3. 임베딩을 이용한 단어 표현

- Semantic Embeddings : Word2Vec

- Google 뉴스 데이터셋에 대하여 사전 학습된 Word2Vec 임베딩을 실험하기 위하여, gensim 라이브러리를 사용한다.

```
import gensim.downloader as api  
w2v = api.load('word2vec-google-news-300')
```

- most_similar 메서드를 이용하여 "neural" 단어와 유사한 단어를 검색

```
for w,p in w2v.most_similar('neural'):  
    print(f"{w} -> {p}")
```



```
neuronal -> 0.780479907989502  
neurons -> 0.7326500415802002  
neural_circuits -> 0.7252851128578186  
neuron -> 0.7174385190010071  
cortical -> 0.6941086053848267  
brain_circuitry -> 0.6923245787620544  
synaptic -> 0.6699119210243225  
neural_circuitry -> 0.6638563275337219  
neurochemical -> 0.6555314064025879  
neuronal_activity -> 0.6531826257705688
```

03. Introduction to NLP with PyTorch

- 3. 임베딩을 이용한 단어 표현

- Semantic Embeddings : Word2Vec

- word_vec 메서드를 이용하여 "play" 단어의 임베딩 벡터 일부 추출

```
w2v.word_vec('play')[:20]
```

```
array([ 0.01226807,  0.06225586,  0.10693359,  0.05810547,  0.23828125,  
        0.03686523,  0.05151367, -0.20703125,  0.01989746,  0.10058594,  
       -0.03759766, -0.1015625 , -0.15820312, -0.08105469, -0.0390625 ,  
       -0.05053711,  0.16015625,  0.2578125 ,  0.10058594, -0.25976562],  
      dtype=float32)
```

- "king"과 "woman" 단어와 의미가 유사하며, "man" 단어와 의미가 다른 단어 추출

```
w2v.most_similar(positive=['king', 'woman'], negative=['man'])[0]
```

```
('queen', 0.7118192911148071)
```

03. Introduction to NLP with PyTorch

- 3. 임베딩을 이용한 단어 표현

- 파이토치에서 사전학습된 임베딩 사용하기
 - 사전 학습된 임베딩과 예시에서 사용할 텍스트 코퍼스와 일치하는 지 확인 및 전처리하는 과정

```
embed_size = len(w2v.get_vector('hello'))
print(f'Embedding size: {embed_size}')

net = EmbedClassifier(vocab_size, embed_size, len(classes))

print('Populating matrix, this will take some time...', end='')
found, not_found = 0, 0
for i, w in enumerate(vocab.itos):
    try:
        net.embedding.weight[i].data = torch.tensor(w2v.get_vector(w))
        found += 1
    except:
        net.embedding.weight[i].data = torch.normal(0.0, 1.0, (embed_size,))
        not_found += 1

print(f'Done, found {found} words, {not_found} words missing')
net = net.to(device)

train_epoch_emb(net, train_loader, lr=4, epoch_size=25000)
```

03. Introduction to NLP with PyTorch

- 3. 임베딩을 이용한 단어 표현

- 파이토치에서 사전학습된 임베딩 사용하기
 - 서로 다른 사전(vocabulary) 문제로 인하여 아래와 같이 해결
 - 현재 텍스트 사전(vocabulary)으로 word2vec 모델 재학습
 - 사전학습된 word2vec 모델로부터 사전을 불러오기
 - GloVe 기반 사전을 torchtext 패키지를 이용하여 인스턴스화
 - 불러온 사전에서 "kind-man" + "woman" = "queen" 을 나타내기 위한 임베딩 조작 예시

```
vocab = torchtext.vocab.GloVe(name='6B', dim=50)

# get the vector corresponding to kind-man+woman
qvec = vocab.vectors[vocab.stoi['king']] - vocab.vectors[vocab.stoi['man']] + 1.3 * vocab.vectors[vocab.stoi['woman']]
# find the index of the closest embedding vector
d = torch.sum((vocab.vectors - qvec)**2, dim=1)
min_idx = torch.argmax(d)
# find the corresponding word
vocab.itos[min_idx]
```

03. Introduction to NLP with PyTorch

• 3. 임베딩을 이용한 단어 표현

- 파이토치에서 사전학습된 임베딩 사용하기
 - 모델 학습을 위한 데이터셋을 준비하기 위하여 offset 벡터를 만드는 offsetify 함수 정의
 - ➔ GloVe 기반 사전 학습된 임베딩 vocab을 사용
 - 정의한 offsetify 함수를 collate_fn으로 정의하여 학습 데이터셋에 대한 DataLoader 생성

```
def offsetify(b):  
    # first, compute data tensor from all sequences  
    x = [torch.tensor(encode(t[1], vocab=vocab)) for t in b] # pass the instance of vocab to encode function!  
    # now, compute the offsets by accumulating the tensor of sequence lengths  
    o = [0] + [len(t) for t in x]  
    o = torch.tensor(o[:-1]).cumsum(dim=0)  
    return (  
        torch.LongTensor([t[0]-1 for t in b]), # labels  
        torch.cat(x), # text  
        o  
    )  
  
net = EmbedClassifier(len(vocab), len(vocab.vectors[0]), len(classes))  
net.embedding.weight.data = vocab.vectors  
net = net.to(device)  
  
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=16, collate_fn=offsetify, shuffle=True)  
train_epoch_emb(net, train_loader, lr=4, epoch_size=25000)
```

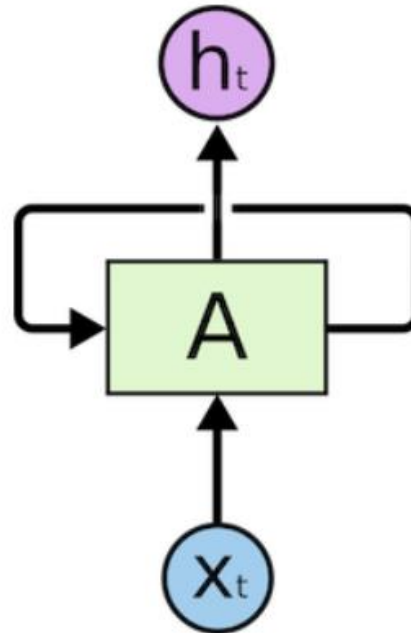
03. Introduction to NLP with PyTorch

- 4. RNN를 이용하여 패턴 탐지

- RNN(Recurrent Neural Networks)

- 기존에 간단한 선형 분류기 모델은 단어의 순서를 반영하여 모델링할 수 없다는 한계점
- 텍스트 시퀀스의 패턴을 탐지하기 위해서는 RNN이라는 또다른 네트워크 구조를 사용한다.

※ RNN은 입력과 출력을 시퀀스 단위로 처리하는 모델로서 은닉층의 노드에서 활성화 함수를 통해 나온 결과값을 출력층 방향으로 보내면서, 다시 은닉층 노드의 다음 계산의 입력으로 보내는 특징을 갖고있습니다.



03. Introduction to NLP with PyTorch

- 4. RNN를 이용하여 패턴 탐지

- 간단한 RNN(Recurrent Neural Networks) 분류기
 - torch.nn.RNN 메서드를 이용하여 순환 층(RNN Layer) 생성
 - 이전 섹션에서 정의한 임베딩 레이어를 이용하여 입력 사전(vocabulary)의 차원을 낮추고 순환 층 적용
 - 순환 층을 거치고 나면 2가지 다음과 같은 출력이 생성
 - 1) x : 각 출력에서 RNN 셀에서 나온 출력 시퀀스
 - 2) h : 시퀀스의 마지막 요소에 대한 은닉 층

```
class RNNClassifier(torch.nn.Module):  
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_class):  
        super().__init__()  
        self.hidden_dim = hidden_dim  
        self.embedding = torch.nn.Embedding(vocab_size, embed_dim)  
        self.rnn = torch.nn.RNN(embed_dim, hidden_dim, batch_first=True)  
        self.fc = torch.nn.Linear(hidden_dim, num_class)  
  
    def forward(self, x):  
        batch_size = x.size(0)  
        x = self.embedding(x)  
        x, h = self.rnn(x)  
        return self.fc(x.mean(dim=1))
```

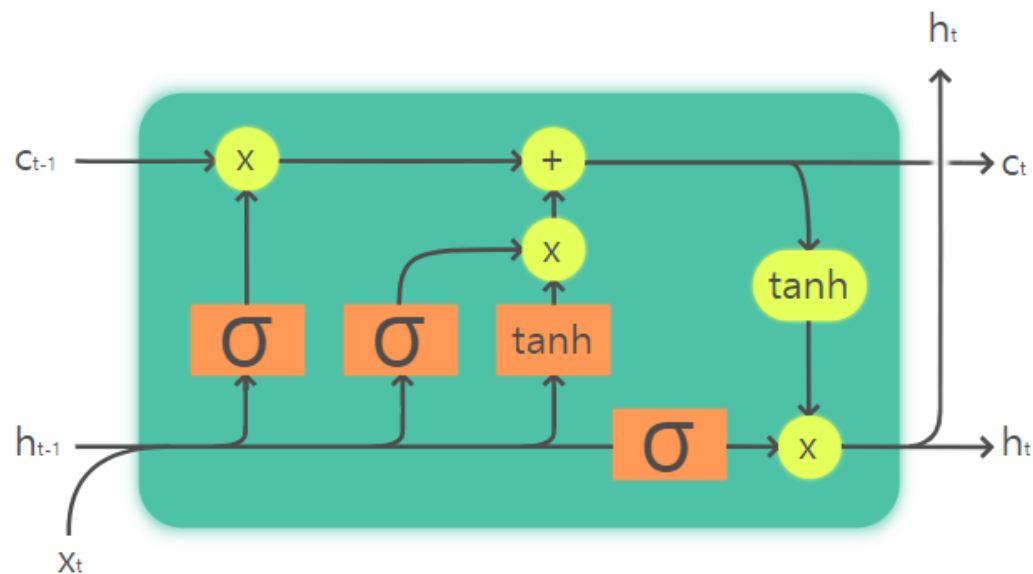
```
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=16, collate_fn=padify, shuffle=True)  
net = RNNClassifier(vocab_size, 64, 32, len(classes)).to(device)  
train_epoch(net, train_loader, lr=0.001)
```

03. Introduction to NLP with PyTorch

- 4. RNN를 이용하여 패턴 탐지

- Long Short Term Memory(LSTM)

- RNN의 주요 문제 중 하나인 기울기 소실(vanishing gradient)를 해결하기 위한 네트워크 중 하나
- LSTM은 RNN과 유사한 방식으로 구성되지만, 계층 간에 전달되는 상태가 단기 상태(h)와 장기 상태(c)로 구성



Legend:

Layer



Pointwise op



Copy



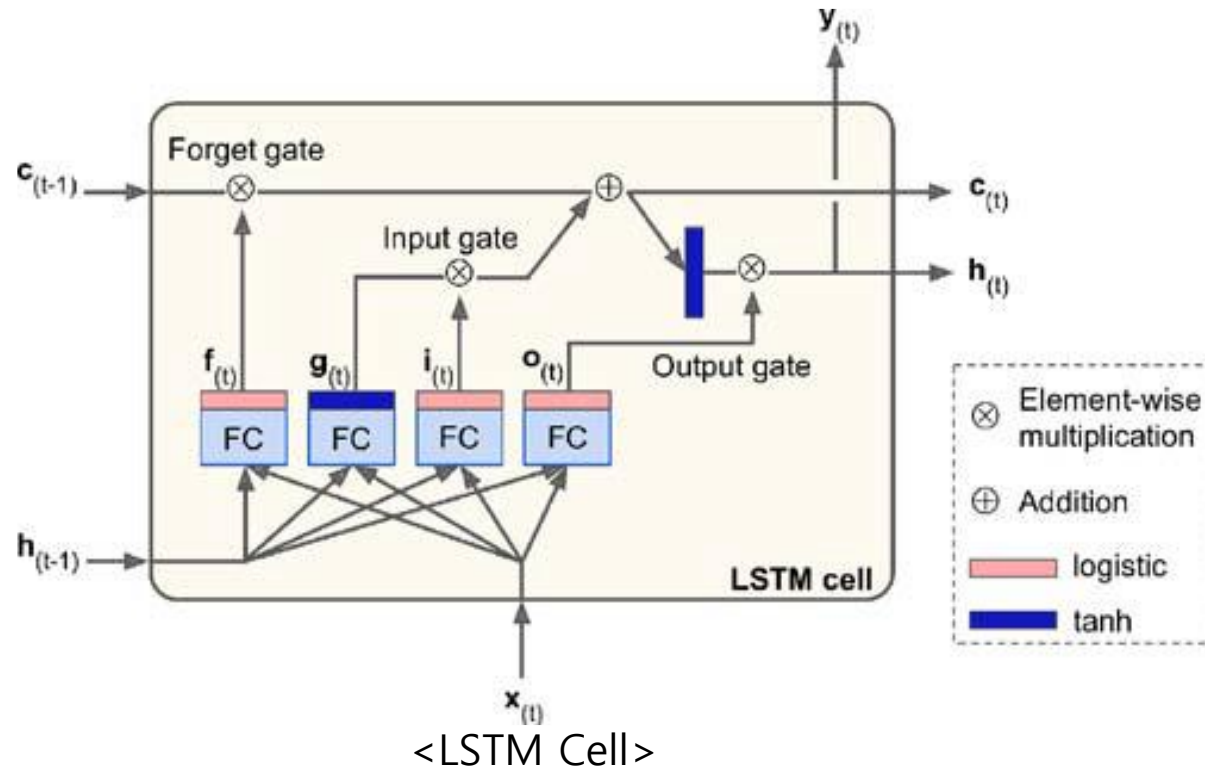
03. Introduction to NLP with PyTorch

4. RNN를 이용하여 패턴 탐지

• Long Short Term Memory(LSTM)

➔ RNN에서 순서대로 정보를 입력받을 때 정보를 선택적으로 사용할 수 있도록 셀 내의 구조를 변형

- 1) Input gate : 주 층($g(t)$)의 어느 부분이 삭제되어야 하는지 $i(t)$ 에서 제어
- 2) Forget gate : 장기 상태($c_{(t-1)}$)의 어느 부분이 삭제되어야 하는지 $f(t)$ 에서 제어
- 3) Output gate : 장기 상태($c_{(t-1)}$)의 어느 부분을 읽어서 다음 타임 스텝의 $h(t)$ 와 $y(t)$ 로 출력해야 하는지 $o(t)$ 가 제어



$h_{(t)}$: 단기 상태(short-term state)

$c_{(t)}$: 장기 상태(long-term state)

$$i_{(t)} = \sigma(W_{xi}^T x_{(t)} + W_{hi}^T h_{(t-1)} + b_i)$$

$$f_{(t)} = \sigma(W_{xf}^T x_{(t)} + W_{hf}^T h_{(t-1)} + b_f)$$

$$o_{(t)} = \sigma(W_{xo}^T x_{(t)} + W_{ho}^T h_{(t-1)} + b_o)$$

$$g_{(t)} = \tanh(W_{xg}^T x_{(t)} + W_{hg}^T h_{(t-1)} + b_g)$$

$$c_{(t)} = f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)}$$

$$y_{(t)} = h_{(t)} = o_{(t)} \otimes \tanh(c_{(t)})$$

03. Introduction to NLP with PyTorch

- 4. RNN를 이용하여 패턴 탐지

- Long Short Term Memory(LSTM)

- torch.nn.LSTM 메서드를 통해 전체 LSTM 계층을 제공한다.

※ LSTM은 훈련 속도가 느리고 훈련 초기에는 정확도가 크게 높지 않을 수 있습니다.

```
class LSTMClassifier(torch.nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_class):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.embedding = torch.nn.Embedding(vocab_size, embed_dim)
        self.embedding.weight.data = torch.randn_like(self.embedding.weight.data) * 0.5
        self.rnn = torch.nn.LSTM(embed_dim, hidden_dim, batch_first=True)
        self.fc = torch.nn.Linear(hidden_dim, num_class)

    def forward(self, x):
        batch_size = x.size(0)
        x = self.embedding(x)
        x, (h, c) = self.rnn(x)
        return self.fc(h[-1])

net = LSTMClassifier(vocab_size, 64, 32, len(classes)).to(device)
train_epoch(net, train_loader, lr=0.001)
```

03. Introduction to NLP with PyTorch

• 4. RNN를 이용하여 패턴 탐지

• Packed sequences

- 시퀀스 데이터를 처리할 때, 매 배치(batch)마다 고정된 문장의 길이로 만들어주기 위해서 <pad>토큰을 넣어야 한다.
- 아래 그림의 파란색 영역은 <pad> 토큰이다. 이러한 연산을 하게 되면, 쓸모없는 <pad>토큰까지 연산을 하게 된다.
- 따라서 <pad>를 계산 안하고 효율적으로 진행하기 위해 다음과 같은 조건을 만족하며 병렬처리를 하려고 한다.
 - 1) RNN의 은닉 층이 이전 타임스텝에 의존해서 최대한 많은 토큰을 병렬적으로 처리해야 한다.
 - 2) 각 문장의 마지막 토큰이 마지막 타임스텝에서 계산을 멈춰야 한다.

| | | | | | |
|------|------|-----|------|---|---------|
| I | love | Mom | ' | s | cooking |
| I | love | you | too | ! | |
| No | way | | | | |
| This | is | the | shit | | |
| Yes | | | | | |

| | | | | | |
|------|------|-----|------|---|---------|
| I | love | Mom | ' | s | cooking |
| I | love | you | too | ! | |
| No | way | | | | |
| This | is | the | shit | | |
| Yes | | | | | |

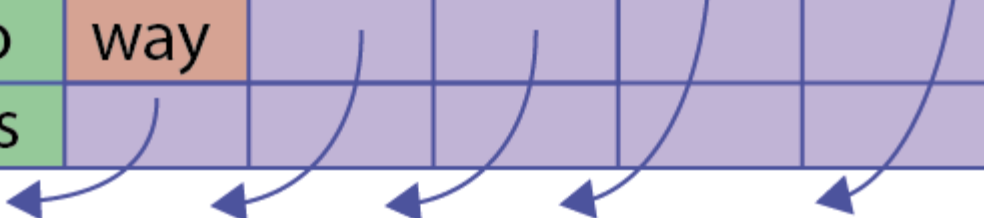
03. Introduction to NLP with PyTorch

• 4. RNN를 이용하여 패턴 탐지

- Packed sequences

- 즉, 컴퓨터로 하여금 각 타임스텝마다 일련의 단어를 처리해야 하므로 각 배치 내에 문장의 길이를 기준으로 정렬한 후, 하나의 통합된 배치로 만들어 주는 과정을 Packed sequences라 한다.
- Packed sequence의 장점은 <pad>토큰을 계산 안하기 때문에 더 빠른 연산 처리를 할 수 있다.

| | | | | | |
|------|------|-----|------|---|---------|
| I | love | Mom | ' | s | cooking |
| I | love | you | too | ! | |
| This | is | the | shit | | |
| No | way | | | | |
| Yes | | | | | |



data



batch_sizes [5, 4, 3, 3, 2, 1]

03. Introduction to NLP with PyTorch

- 4. RNN를 이용하여 패턴 탐지

- Packed sequences

- packed sequences를 생성하기 위해서 벡터의 길이 값을 네트워크에 전달해야 하므로 retur에 torch.tensor(len_seq) 추가

```
def pad_length(b):  
    # build vectorized sequence  
    v = [encode(x[1]) for x in b]  
    # compute max length of a sequence in this minibatch and length sequence itself  
    len_seq = list(map(len,v))  
    l = max(len_seq)  
    return ( # tuple of three tensors - labels, padded features, length sequence  
            torch.LongTensor([t[0]-1 for t in b]),  
            torch.stack([torch.nn.functional.pad(torch.tensor(t), (0, l-len(t)), mode='constant', value=0) for t in v]),  
            torch.tensor(len_seq)  
            )  
  
train_loader_len = torch.utils.data.DataLoader(train_dataset, batch_size=16, collate_fn=pad_length, shuffle=True)
```

03. Introduction to NLP with PyTorch

- 4. RNN를 이용하여 패턴 탐지

- Packed sequences

- torch.nn.utils.rnn.pack_padded_sequence 메서드를 사용하여 packed sequence를 사용할 수 있다.
- 이를 통해 생성된 packed sequence는 모든 RNN 계층(RNN, LSTM, GRU 등)의 입력으로 제공된다.
- RNN 계층의 출력값에 torch.nn.utils.rnn.pack_padded_sequence 메서드를 사용하여 디코딩 수행

```
class LSTMpackClassifier(torch.nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_class):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.embedding = torch.nn.Embedding(vocab_size, embed_dim)
        self.embedding.weight.data = torch.randn_like(self.embedding.weight.data) * 0.5
        self.rnn = torch.nn.LSTM(embed_dim, hidden_dim, batch_first=True)
        self.fc = torch.nn.Linear(hidden_dim, num_class)

    def forward(self, x, lengths):
        batch_size = x.size(0)
        x = self.embedding(x)
        pad_x = torch.nn.utils.rnn.pack_padded_sequence(x, lengths, batch_first=True, enforce_sorted=False)
        pad_x, (h, c) = self.rnn(pad_x)
        x, _ = torch.nn.utils.rnn.pad_packed_sequence(pad_x, batch_first=True)
        return self.fc(h[-1])
```

03. Introduction to NLP with PyTorch

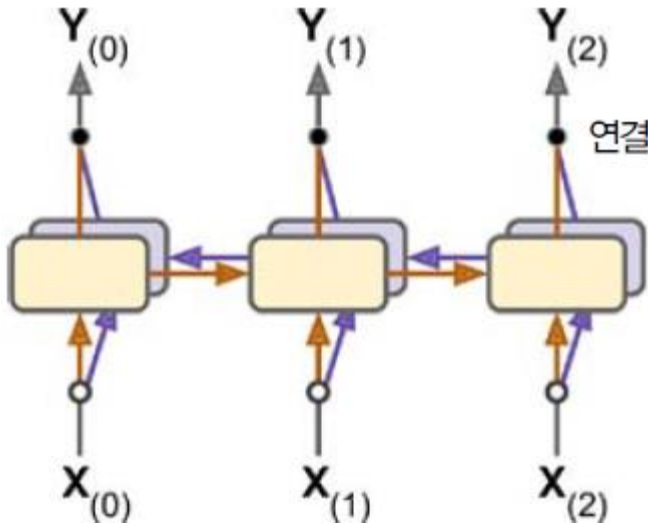
• 4. RNN를 이용하여 패턴 탐지

- 양방향 RNN(Bidirectional RNNs)

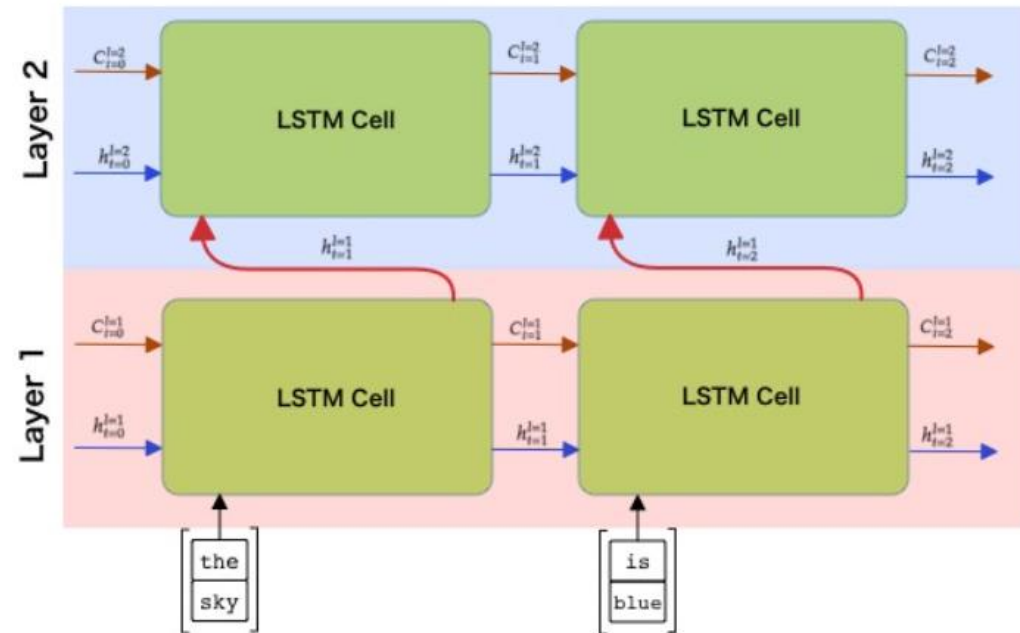
- 이전 예시에서 모든 순환 신경망은 입력 시퀀스의 시작부터 끝까지 한 방향으로 작동했다. 하지만 과거와 현재의 입력만 보고 출력을 생성하는 일반적인 순환층의 구조는 자연어 처리 작업에는 맞지 않음
- RNN/LSTM/GRU 함수의 파라미터로 `bidirectional=True`로 설정하여 양방향 RNN 생성 가능하다.

- 다층 순환 신경망(Multilayer RNNs)

- RNN 셀을 추가하여 더 높은 수준의 패턴을 추출



<양방향 순환층>



<다층 순환 층>

03. Introduction to NLP with PyTorch

• 5. 순환 네트워크에서 텍스트 생성

• 생성 네트워크(Generative network)

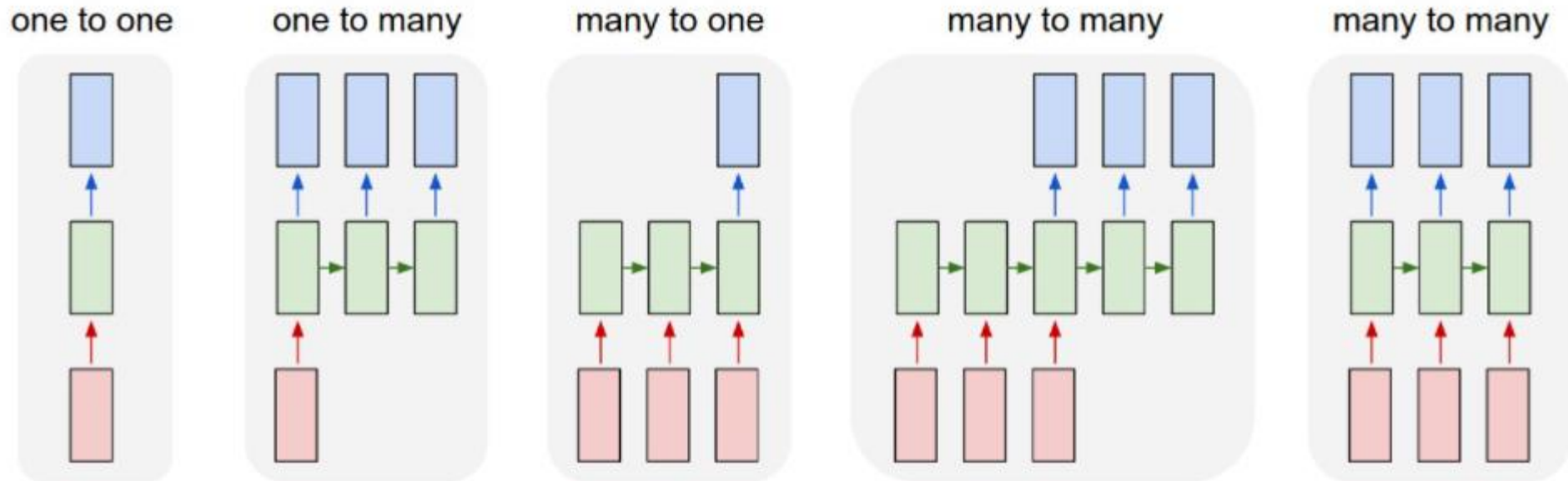
- RNN 및 LSTM과 같은 모델은 텍스트 생성, 기계 번역, 이미지 캡션과 같은 생성 작업(generative task)에 사용된다.
- 입력과 출력 시퀀스에 따라 아래와 같이 다른 네트워크 구조를 갖는다.

1) one-to-one : 하나의 입력 노드에 하나의 출력 노드로 구성된 기존의 신경망

2) one-to-many : 이미지(또는 CNN의 출력)를 입력하여 이미지에 대한 캡션을 만들 때 활용

3) many-to-one : 영화 리뷰에 있는 연속된 단어를 주입하여 네트워크는 감성 점수를 출력

4) many-to-many : 시퀀스-투-시퀀스 네트워크라고도 불리며 기계 번역 작업에 활용



03. Introduction to NLP with PyTorch

- 5. 순환 네트워크에서 텍스트 생성

- 문자 사전(character vocabulary) 구축하기
 - 이전 섹션에서의 단어 수준의 토큰화 작업이 아닌 문자 수준의 토큰화 작업을 위하여 새로운 tokenizer 정의

```
def char_tokenizer(words):  
    return list(words) # [word for word in words]  
  
counter = collections.Counter()  
for (label, line) in train_dataset:  
    counter.update(char_tokenizer(line))  
vocab = torchtext.vocab.Vocab(counter)  
  
vocab_size = len(vocab)  
print(f"Vocabulary size = {vocab_size}")  
print(f"Encoding of 'a' is {vocab.stoi['a']}")  
print(f"Character with code 13 is {vocab.itos[13]}")
```

Vocabulary size = 84

Encoding of 'a' is 4

Character with code 13 is h

03. Introduction to NLP with PyTorch

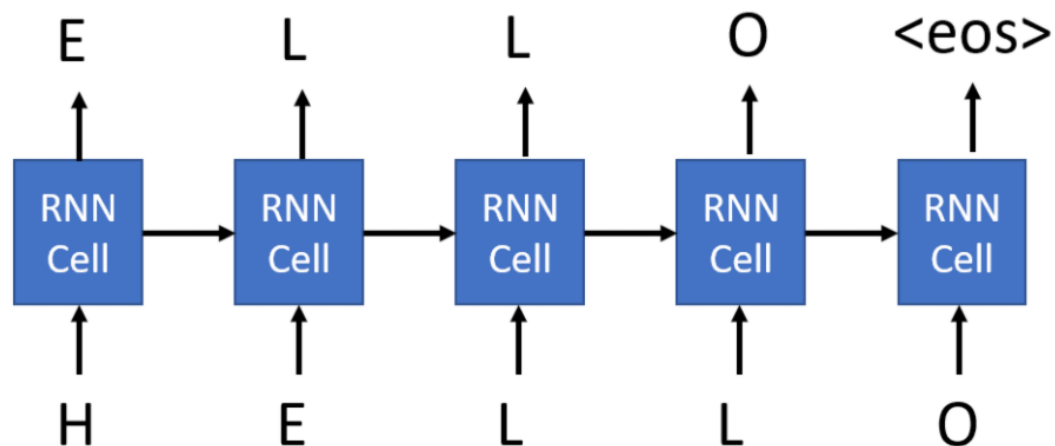
• 5. 순환 네트워크에서 텍스트 생성

- generative RNN 학습하기

- 각 학습 스텝마다 특정 nchars 길이의 단어 시퀀스를 이용하여, 각 입력 문자에 대한 다음 출력 문자를 생성

※ <eos> : end of sequences

- get_batch 함수를 정의하여 특정 길이의 뉴스 텍스트로부터 미니배치를 생성



```
nchars = 100

def get_batch(s, nchars=nchars):
    ins = torch.zeros(len(s)-nchars, nchars, dtype=torch.long, device=device)
    outs = torch.zeros(len(s)-nchars, nchars, dtype=torch.long, device=device)
    for i in range(len(s)-nchars):
        ins[i] = enc(s[i:i+nchars])
        outs[i] = enc(s[i+1:i+nchars+1])
    return ins, outs

get_batch(train_dataset[0][1])
```

03. Introduction to NLP with PyTorch

- 5. 순환 네트워크에서 텍스트 생성

- generative RNN 학습하기
 - LSTM 계층을 사용하여 생성 모델 구축
 - 모델은 문자(character)를 입력으로 사용하므로 임베딩이 필요하지 않고, 원-핫 인코딩 벡터로 LSTM 계층에 주입

```
class LSTMGenerator(torch.nn.Module):
    def __init__(self, vocab_size, hidden_dim):
        super().__init__()
        self.rnn = torch.nn.LSTM(vocab_size, hidden_dim, batch_first=True)
        self.fc = torch.nn.Linear(hidden_dim, vocab_size)

    def forward(self, x, s=None):
        x = torch.nn.functional.one_hot(x, vocab_size).to(torch.float32)
        x, s = self.rnn(x, s)
        return self.fc(x), s
```

- generate 함수로 "today"로 시작하고 길이 100의 텍스트 생성하기 위하여 정의

```
def generate(net, size=100, start='today '):
    chars = list(start)
    out, s = net(enc(chars).view(1, -1).to(device))
    for i in range(size):
        nc = torch.argmax(out[0][-1])
        chars.append(vocab.itos[nc])
        out, s = net(nc.view(1, -1), s)
    return ''.join(chars)
```

03. Introduction to NLP with PyTorch

- 5. 순환 네트워크에서 텍스트 생성

- generative RNN 학습하기
 - 다음과 같은 훈련 루프를 통하여 학습 시작
 - 학습 epoch마다 loss값과 생성한 텍스트를 함께 출력

```
net = LSTMGenerator(vocab_size,64).to(device)

samples_to_train = 10000
optimizer = torch.optim.Adam(net.parameters(),0.01)
loss_fn = torch.nn.CrossEntropyLoss()
net.train()
for i,x in enumerate(train_dataset):
    # x[0] is class label, x[1] is text
    if len(x[1])-nchars<10:
        continue
    samples_to_train-=1
    if not samples_to_train: break
    text_in, text_out = get_batch(x[1])
    optimizer.zero_grad()
    out,s = net(text_in)
    loss = torch.nn.functional.cross_entropy(out.view(-1,vocab_size),text_out.flatten()) #cross_entropy(out,labels)
    loss.backward()
    optimizer.step()
    if i%1000==0:
        print(f"Current loss = {loss.item()}")
        print(generate(net))
```

Current loss = 1.52818763256073

today and the company of the company of the company of the company of the company of the company of the co

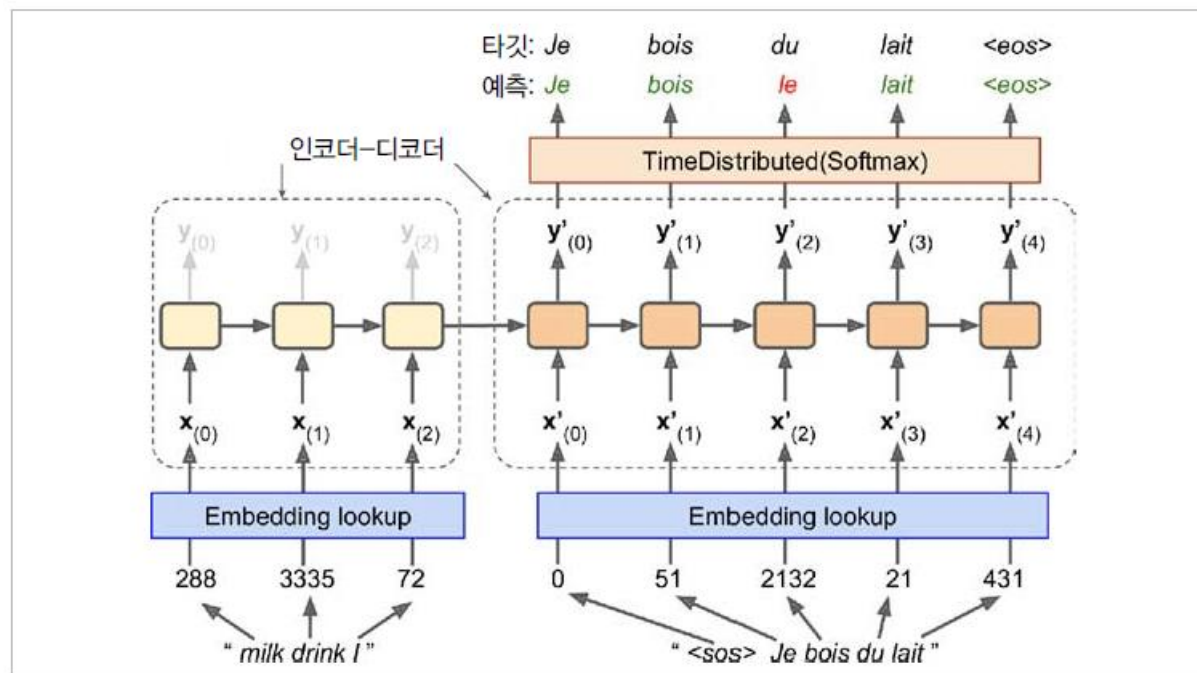
Current loss = 1.5444810390472412

today and thecounters to the first the counters to the first the counters to the first the counters to th

03. Introduction to NLP with PyTorch

• 6. 어텐션 모델과 트랜스포머

- 기계 번역(machine translation)을 위한 인코더-디코더 모델
 - 영어 문장을 프랑스어로 번역하는 간단한 신경망 기계 번역 모델
 - 인코더 : 영어 문장("I drink milk") / 디코더 : 번역된 프랑스어("Je bois du lait")
 - 문장 임베딩 : 문자열을 입력으로 받아 하나의 벡터로 인코딩
- ※ 영어 문장 "I drink milk"를 번역하기 위하여 거꾸로 뒤집어 "milk drink I"로 주입



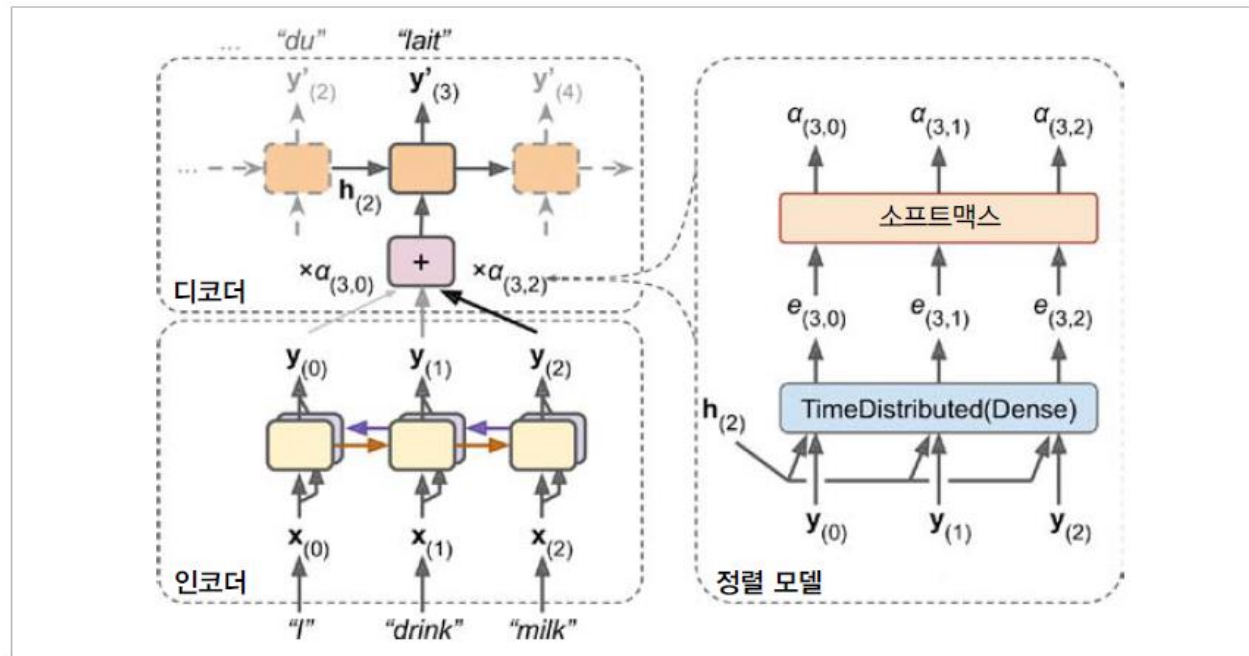
<간단한 신경망 기계 번역 모델>

03. Introduction to NLP with PyTorch

• 6. 어텐션 모델과 트랜스포머

• 어텐션 메커니즘(Attention mechanism) – **Additive Attention**

- 정렬 모델(Alignment model) 또는 어텐션 층(Attention layer)을 통해 단어 "milk"에 집중할 수 있는 가중치 생성
- 인코더의 전체 은닉층 $y_{(i)}$ 과 디코더의 이전 은닉층 $h_{(2)}$ 의 유사도 $e_{(3,i)}$ 계산 = **Similarity function**
- 계산된 유사도 $e_{(3,i)}$ 를 소프트맥스 층을 통과하여 인코더 출력에 대한 최종 가중치 $a_{(3,i)}$ 계산 = **Attention Score**
- 계산된 가중치 $a_{(3,i)}$ 와 인코더의 전체 은닉층 $y_{(i)}$ 의 가중치 합을 이용하여 Feature 표현 : $\sum y_{(i)} * a_{(3,i)}$



<어텐션 모델의 인코더-디코더 네트워크를 사용한 신경망 기계 번역>

03. Introduction to NLP with PyTorch

• 6. 어텐션 모델과 트랜스포머

- 트랜스포머(Transformer) 모델

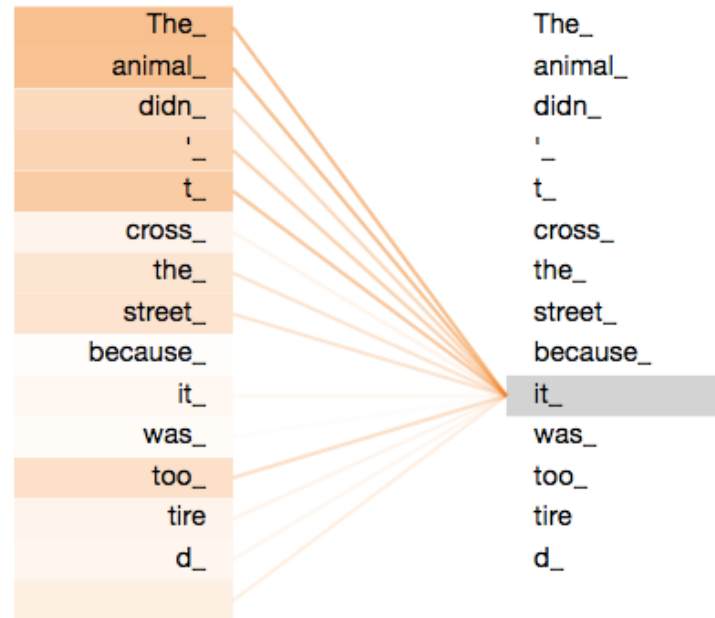
- RNN 기반의 어텐션 메커니즘 문제점

- 1) 시퀀셜 데이터에서의 순차적 학습으로 인해 병렬 처리 X ➔ 연산 속도 저하
- 2) 입력 길이가 길어질수록 Long-term dependency

- Self-Attention : 각 단어와 동일한 문장에 있는 다른 단어의 관계를 인코딩하는 방법

➔ RNN(또는 CNN) 구조를 사용하지 않고 Attention만을 사용하여 Feature representation ➔ Transformer

예시) "The animal didn't cross the street because **it** was too tired."



<예시 문장의 셀프-어텐션 가중치 시각화>

03. Introduction to NLP with PyTorch

• 6. 어텐션 모델과 트랜스포머

• 트랜스포머(Transformer) 모델

- 시퀀셜 데이터에 대하여 RNN 구조를 사용하지 않고 어텐션만을 사용하는 모델
→ 훨씬 빠르게 훈련할 수 있고 병렬화하기 쉬운 점이 장점

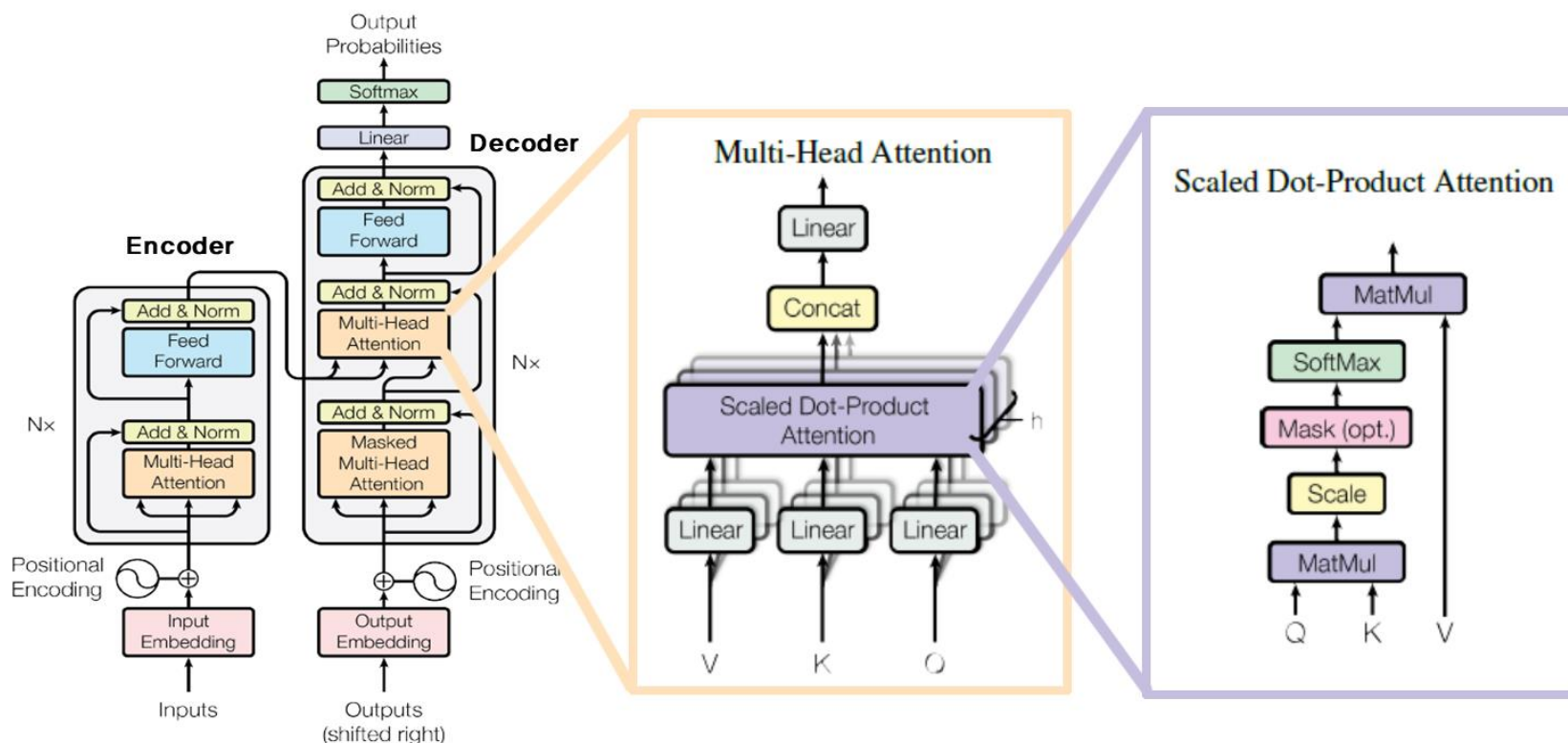


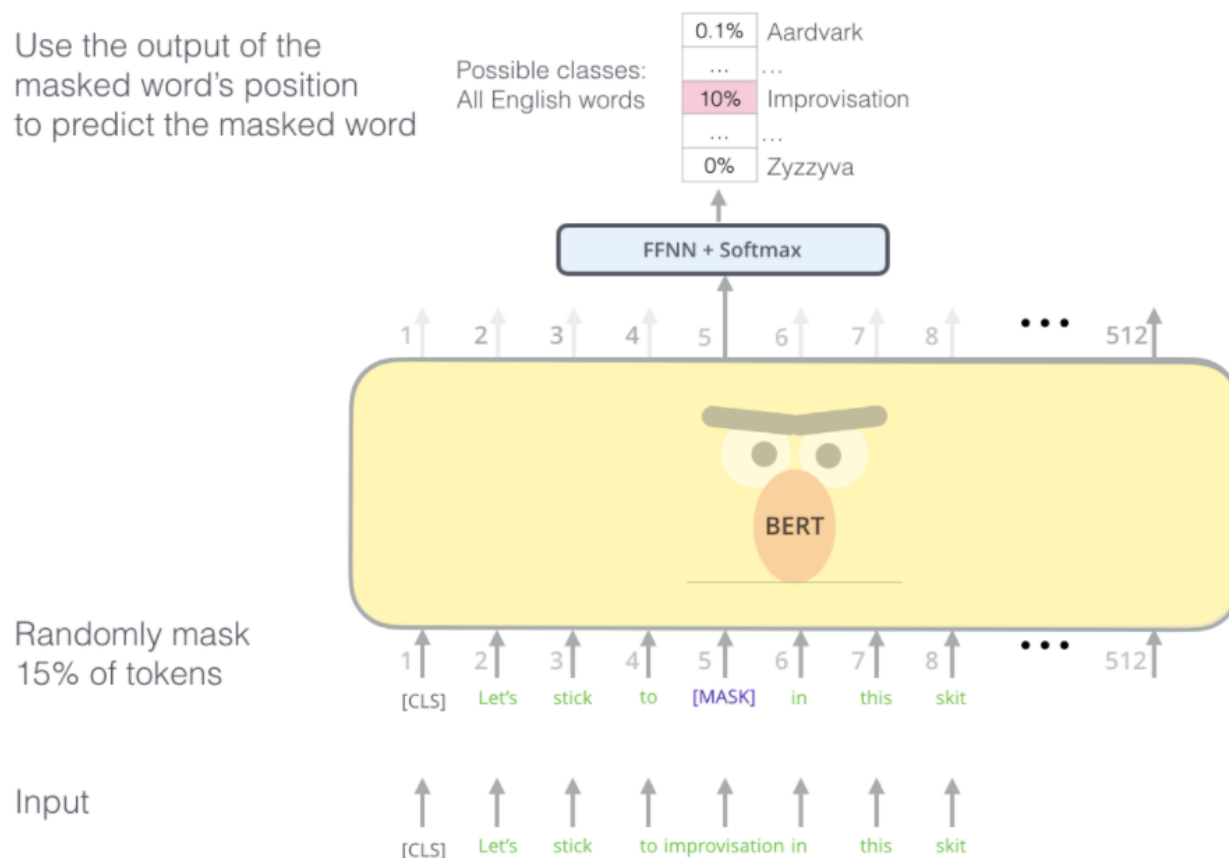
Figure 1: The Transformer - model architecture.

<Transformer 구조>

03. Introduction to NLP with PyTorch

• 6. 어텐션 모델과 트랜스포머

- BERT(Bidirectional Encoder Representations from Transformers)
 - 모델은 먼저 비지도 학습을 사용하여 텍스트 데이터의 대규모 말뭉치(Wikipedia + 책)에 대해 사전 훈련된다.
 - 이를 통해 모델은 상당한 수준의 언어를 이해하고, fine-tuning을 사용하여 다른 데이터셋과 함께 활용할 수 있다. 이 과정을 전이 학습(Transfer learning)이라고 한다.



03. Introduction to NLP with PyTorch

• 6. 어텐션 모델과 트랜스포머

- BERT(Bidirectional Encoder Representations from Transformers)
 - 대규모 텍스트 데이터에서 자기 지도 사전훈련을 이용
 - GPT와 비슷한 구조지만 Transformer의 Encoder처럼 마스킹이 없는 Multi-Head Attention 사용 = 양방향성(Bidirectional)
- BERT는 모델의 강점으로 2가지 사전훈련 작업을 제안
 - 1) 마스크드 언어 모델(Masked language model, MLM)
 - 문장에 있는 각 단어는 순서에 상관없이 15%의 확률로 마스킹하고 모델은 마스킹한 단어를 예측하도록 훈련됩니다
 - 예시 : ""She had fun at the birthday party"라는 문장에서
"She <mask> fun at the <mask> party" → "had" 와 "birthday"를 예측
 - 2) 다음 문장 예측(Next sentence prediction, NSP)
 - 두 문장이 연속적인지 아닌지를 예측하도록 모델을 훈련
 - 예시 : "The dog sleeps" → "It snores loudly" (O)
"The dog sleeps" → "The Earth orbits the Sun" (X)

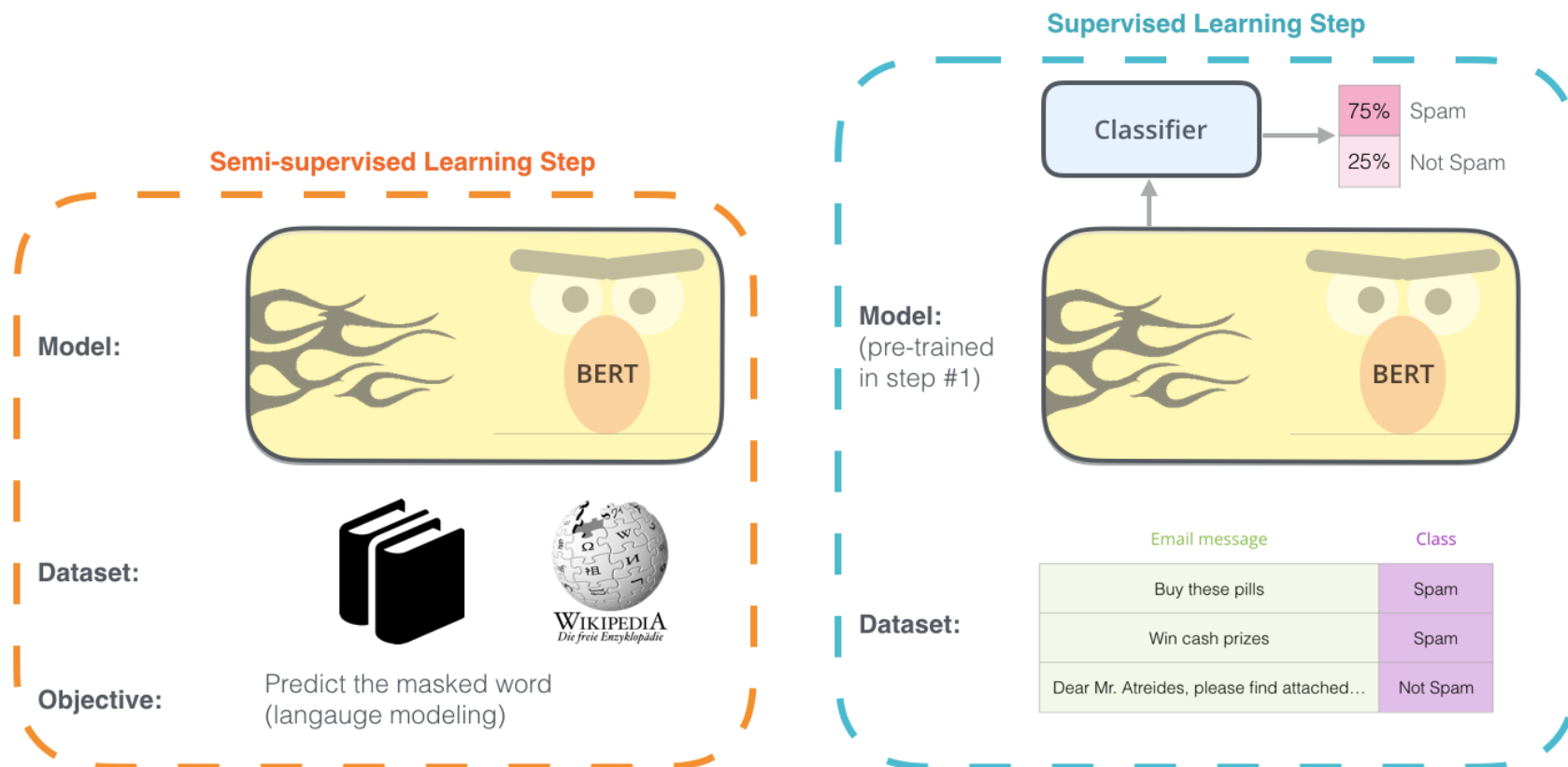
03. Introduction to NLP with PyTorch

• 6. 어텐션 모델과 트랜스포머

- BERT : The two steps of how BERT is developed

Step 1) Semi-supervised Learning : 책, 위키피디아 등 unlabeled text로 사전학습된 모델

Step 2) Supervised Learning : 특정 Task에 대한 labeled text 데이터로 지도학습에서 fine-tuning 수행



03. Introduction to NLP with PyTorch

• 6. 어텐션 모델과 트랜스포머

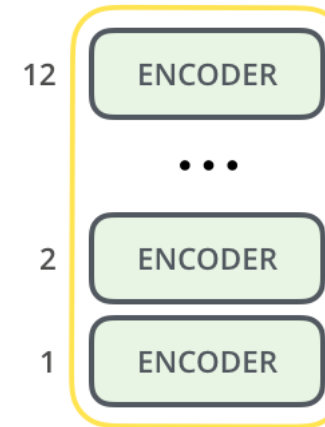
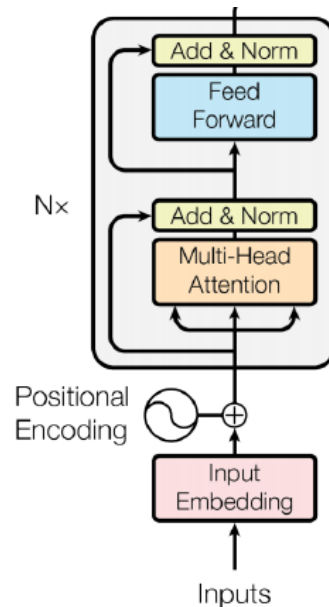
- BERT: Model Architecture - Multi-layer bidirectional Transformer encoder
 - L : number of layers (Transformer block)
 - H : hidden size
 - A : number of self attention heads

- BERT BASE

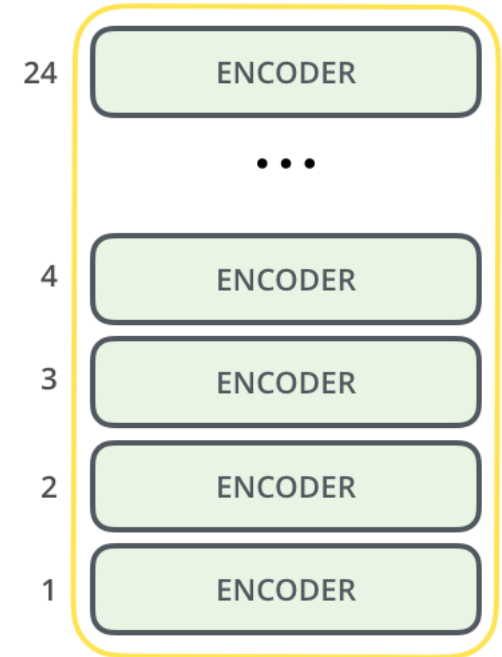
- L = 12, H = 768, A = 12
- Total parameters = 110M
- same model size as OpenAI GPT

- BERT LARGE

- L = 24, H = 1,024, A = 16
- Total parameters = 340M



BERT_{BASE}



BERT_{LARGE}

03. Introduction to NLP with PyTorch

• 6. 어텐션 모델과 트랜스포머

- BERT를 사용하여 텍스트 분류 모델 구축하기
 - 사전 훈련된 BERT 모델을 사용하여 AG News 데이터셋을 분류
 - transformers.BertTokenizer.from_pretrained 함수를 이용하여 BERT 모델과 사전 훈련된 tokenizer 로드
- ※ tokenizer 객체는 텍스트 인코딩 기능을 포함하고 있다.

```
import torch
import torchtext
from torch.nn import *
import transformers

train_dataset, test_dataset, classes, vocab = load_dataset()
vocab_len = len(vocab)

# To load the model from Internet repository using model name.
# Use this if you are running from your own copy of the notebooks
bert_model = 'bert-base-uncased'

# To load the model from the directory on disk. Use this for Microsoft Learn module, because we have
# prepared all required files for you.
bert_model = './bert'

tokenizer = transformers.BertTokenizer.from_pretrained(bert_model)

MAX_SEQ_LEN = 128
PAD_INDEX = tokenizer.convert_tokens_to_ids(tokenizer.pad_token)
UNK_INDEX = tokenizer.convert_tokens_to_ids(tokenizer.unk_token)
```

03. Introduction to NLP with PyTorch

- 6. 어텐션 모델과 트랜스포머

- BERT를 사용하여 텍스트 분류 모델 구축하기

- BERT는 자체 인코딩 함수를 사용하기 때문에, 이전에 정의한 시퀀스 길이를 동일하게 맞추는 padify 함수와 유사하게 pad_bert 함수를 정의하여 데이터로더 생성

```
def pad_bert(b):  
    # b is the list of tuples of length batch_size  
    # - first element of a tuple = label,  
    # - second = feature (text sequence)  
    # build vectorized sequence  
    v = [tokenizer.encode(x[1]) for x in b]  
    # compute max length of a sequence in this minibatch  
    l = max(map(len, v))  
    return ( # tuple of two tensors - labels and features  
        torch.LongTensor([t[0] for t in b]),  
        torch.stack([torch.nn.functional.pad(torch.tensor(t), (0, l-len(t)), mode='constant', value=0) for t in v])  
    )
```

```
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=8, collate_fn=pad_bert, shuffle=True)  
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=8, collate_fn=pad_bert)
```

03. Introduction to NLP with PyTorch

• 6. 어텐션 모델과 트랜스포머

- BERT를 사용하여 텍스트 분류 모델 학습하기
 - BertForSequenceClassification 패키지를 사용하여 사전 훈련된 BERT 모델(bert-base-uncased) 로드
 - 학습 파라미터를 설정한 후 모델 학습 수행

```
model = transformers.BertForSequenceClassification.from_pretrained(bert_model, num_labels=4).to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=2e-5)

report_freq = 50
iterations = 500 # make this larger to train for longer time!

model.train()

i, c = 0, 0
acc_loss = 0
acc_acc = 0

for labels, texts in train_loader:
    labels = labels.to(device)-1 # get labels in the range 0-3
    texts = texts.to(device)
    loss, out = model(texts, labels=labels)[:2]
    labs = out.argmax(dim=1)
    acc = torch.mean((labs==labels).type(torch.float32))
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    acc_loss += loss
    acc_acc += acc
    i+=1
    c+=1
    if i%report_freq==0:
        print(f"Loss = {acc_loss.item()/c}, Accuracy = {acc_acc.item()/c}")
        c = 0
        acc_loss = 0
        acc_acc = 0
    iterations-=1
    if not iterations:
        break
```

03. Introduction to NLP with PyTorch

- 6. 어텐션 모델과 트랜스포머

- BERT를 사용하여 텍스트 분류 모델 평가하기
 - 테스트 데이터셋(test_loader)를 이용하여 모델 평가 수행
 - model.eval()를 호출하여 모델 평가 모드 전환 필수

```
model.eval()
iterations = 100
acc = 0
i = 0
for labels, texts in test_loader:
    labels = labels.to(device)-1
    texts = texts.to(device)
    _, out = model(texts, labels=labels)[:2]
    labs = out.argmax(dim=1)
    acc += torch.mean((labs==labels).type(torch.float32))
    i+=1
    if i>iterations: break

print(f"Final accuracy: {acc.item()/i}")
```




End of document

Website

<http://www.brframe.com/>

Contact Point

E-mail admin@brframe.com