

# 6장 임베딩

## (도입) 텍스트 벡터화: 원-핫 인코딩, 빈도 벡터화

- 자연어를 모델이 다루려면 **텍스트 벡터화(Text Vectorization)** 가 필요하다.
- 기초적 방식: **원-핫 인코딩(One-Hot Encoding)**, **빈도 벡터화(Count Vectorization)**.
- 원-핫 인코딩: 문서의 각 단어를 고유 **색인**에 매핑하고, 해당 위치만 1로 나머지는 0.
  - 예시(색인 매핑, 표 6.1 원-핫 인코딩 매핑 테이블):  
색인 0: **I**, 1: **like**, 2: **apples**, 3: **bananas**
  - 문장 벡터
    - **I like apples** → [1, 1, 1, 0]
    - **I like bananas** → [1, 1, 0, 1]
- **빈도 벡터화**: 문서에서 단어의 **등장 빈도**로 벡터를 만든다(예: **apples** 가 4번이면 벡터값 4).
- 한계
  - **희소성(sparsity)** 큼(차원↑, 비용↑).
  - **의미 보존 어려움**: 문장의 의미가 유사해도 벡터가 유사하게 나오지 않을 수 있음.  
예: **i scream for ice cream** 과 **ice cream for i scream** → 원-핫/빈도 모두 **동일한** 벡터 결과.
- 이를 보완하기 위해 **워드 임베딩(Word Embedding)** 개념 사용: 단어를 **고정 길이 실수 벡터**로 나타내 의미 관계를 반영.  
고정 임베딩의 한계(다의어/문맥 반영 어려움)로, **동적 임베딩(Dynamic Embedding)**(문맥 의존 신경망 활용)이 소개된다.

## 언어 모델

- **정의**: 입력된 문장으로 **각 문장을 생성할 확률**을 계산하는 모델.  
주어진 문장을 바탕으로 문맥을 이해하고, 문장 구성에 대한 **예측**을 수행.
- **활용**: **자동 번역, 음성 인식, 텍스트 요약** 등.

$P(\text{만나서 반갑습니다}) = 0.081$	$P(\text{서울특별시}) = 0.59$
Input : 안녕하세요 ...	Input : 대한민국 ... 강남구 ...
$P(\text{아니오, 괜찮습니다}) = 0.000001$	$P(\text{아이스 아메리카노}) = 0.000001$

그림 6.1 언어 모델

입력 문장에 대해 다음 단어 후보들의 확률을 산출하는 구조 예시

## 자기회귀 언어 모델

- 정의: 입력된 문장들의 조건부 확률을 이용해 다음에 올 단어를 예측.

이전 단어들의 시퀀스가 주어졌을 때 다음 단어의 확률을 계산.

- 수식 6.1 (언어 모델의 조건부 확률)

$$P(w_t | w_1, w_2, \dots, w_{t-1}) = \frac{P(w_1, w_2, \dots, w_t)}{P(w_1, w_2, \dots, w_{t-1})}$$

- 수식 6.2 (조건부 확률의 연쇄법칙 적용)

$$P(w_t | w_1, w_2, \dots, w_{t-1}) = P(w_1) P(w_2 | w_1) \cdots P(w_t | w_1, w_2, \dots, w_{t-1})$$

(연쇄법칙을 통해 문장 전체 확률을 곱으로 전개하고, 이를 이용해 다음 단어를 예측)

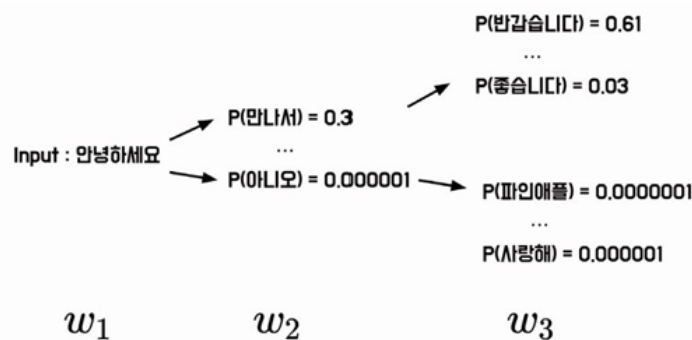


그림 6.2 조건부 확률의 연쇄법칙을 통한 자기회귀 언어 모델 구조

입력 "안녕하세요"에 대해 만나서, 반갑습니다, ... 등 다음 후보의 확률을 단계적으로 계산하는 흐름을 도식화.

## 통계적 언어 모델

- 정의: 언어의 통계적 구조를 이용해 문장/단어의 시퀀스를 분석하고 다음 단어 확률을 예측.

- 일반적 구현: **마르코프 체인(Markov Chain)** 기반 **빈도**(동시/조건부 빈도)로 확률을 추정.

- 예) 말뭉치에서 **안녕하세요** 1000번, 이어서 **만나서** 700번, **반갑습니다** 100번 등장 시:

**수식 6.3 (빈도 기반의 조건부 확률)**

$$P(\text{만나서} \mid \text{안녕하세요}) = \frac{P(\text{안녕하세요 만나서})}{P(\text{안녕하세요})} = \frac{700}{1000}$$

$$P(\text{반갑습니다} \mid \text{안녕하세요}) = \frac{P(\text{안녕하세요 반갑습니다})}{P(\text{안녕하세요})} = \frac{100}{1000}$$

- 한계: 단어 **순서/빈도**만으로 문맥을 충분히 반영하기 어려워 **불완전/부정확** 가능, 관측이 드문 구문에 대한 **데이터 희소성(Data sparsity)** 문제가 발생.

## GPT / BERT

- **GPT**

- Raw Sentence: *GPT는 OpenAI에서 개발한 인과적 언어 모델입니다.*
- Input: *GPT는 OpenAI에서 개발한*
- Prediction-1: *인과적* → Prediction-2: *언어 모델입니다.*

- **BERT**

- Raw Sentence: *BERT는 Google에서 개발한 마스킹된 언어 모델입니다.*
- Input: *Bert는 [MASK]에서 개발한 [MASK] 언어 모델입니다.*
- Prediction: *Google, 마스킹된*
- (*해당 모델의 자세한 내용은 7장에서 다룸 – 본 장에서는 문장 생성 기법과 사전학습된 언어 모델의 개요만 제시*)

## N-gram

- 가장 기초적인 통계적 언어 모델. **연속된 N개 단어 시퀀스(N-gram)**를 단위로 하여 특정 단어 시퀀스의 **등장 확률**을 추정.
- 용어
  - N=1: **유니그램(Unigram)**
  - N=2: **바이그램(Bigram)**
  - N=3: **트라이그램(Trigram)**

Unigram : <u>안녕하세요</u> <u>만나서</u> <u>진심으로</u> <u>반가워요</u>	안녕하세요. 만나서. 진심으로. 반가워요
Bigram : <u>안녕하세요 만나서</u> <u>진심으로 반가워요</u>	안녕하세요 만나서. 만나서 진심으로. 진심으로 반가워요
Trigram : <u>안녕하세요 만나서 진심으로</u> <u>반가워요</u>	안녕하세요 만나서 진심으로. 만나서 진심으로 반가워요

그림 6.3 N이 1, 2, 3인 N-gram

문장 "안녕하세요 만나서 진심으로 반가워요"를 유니그램/바이그램/트라이그램으로 분할한 예시.

- 수식 6.4 (N-gram에서의 조건부 확률)

$$P(w_t \mid w_{t-1}, w_{t-2}, \dots, w_{t-N+1})$$

→ 예측에는 직전 N-1개만 사용하며, N의 크기로 문맥 길이를 조절.

- 예제 6.1 N-gram (파이썬/NLTK)

```
import nltk

def ngrams(sentence, n):
    words = sentence.split()
    ngrams = zip(*(words[i:] for i in range(n)))
    return list(ngrams)

sentence = "안녕하세요 만나서 진심으로 반가워요"
```

```
unigram = ngrams(sentence, 1)
bigram = ngrams(sentence, 2)
trigram = ngrams(sentence, 3)

print(unigram)
print(bigram)
print(trigram)

unigram = nltk.ngrams(sentence.split(), 1)
bigram = nltk.ngrams(sentence.split(), 2)
trigram = nltk.ngrams(sentence.split(), 3)

print(list(unigram))
```

```
print(list(bigram))
print(list(trigram))
```

### 출력 결과

- 유니그램: [('안녕하세요'), ('만나서'), ('진심으로'), ('반가워요')]
- 바이그램: [('안녕하세요', '만나서'), ('만나서', '진심으로'), ('진심으로', '반가워요')]
- 트라이그램: [('안녕하세요', '만나서', '진심으로'), ('만나서', '진심으로', '반가워요')]

(파이썬 구현과 NLTK 함수 모두 동일한 출력)

## TF-IDF

- 정의: *Term Frequency-Inverse Document Frequency*. 문서 내 단어의 중요도를 가중치로 반영하는 방법.

즉, **BoW(Bag-of-Words)** 에 가중치를 부여.

- BoW는 문서를 단어의 집합으로 보고 중복 허용(빈도 기록).  
원-핫은 0/1로만 표시하지만, BoW는 등장 빈도를 고려.

표 6.2 BoW 벡터화

	I	like	this	movie	don't	famous	is
This movie is famous movie	0	0	1	2	0	1	1
I like this movie	1	1	1	1	0	0	0
I don't like this movie	1	1	1	1	1	0	0

예시(문장 집합: [That movie is famous movie], [I like that actor], [I don't like that actor])를 BoW로 벡터화.

BoW만으로는 모든 단어가 동일 가중치라 감성 분류 등에서 한계가 있음.

예: I like this movie vs I don't like this movie 에서 don't 는 분류에 결정적이지만 자주 등장하지 않을 수 있어 무시될 가능성.

### 단어 빈도 (TF)

- 정의: 문서 내에서 특정 단어의 등장 횟수.

수식 6.5 (단어 빈도)

$$TF(t, d) = \text{count}(t, d)$$

- TF가 높다고 항상 중요 단어는 아님(전문 용어나 관용어처럼 **자주 쓰이는 단어**일 수 있음).

문서 길이가 길수록 TF가 커지는 **길이 민감성**도 존재.

## 문서 빈도 (DF)

- **정의:** 전체 문서 집합에서 **몇 개의 문서**에 그 단어가 나타나는지(문서 **출현 횟수**).

### 수식 6.6 (문서 빈도)

$$DF(t, D) = \text{count}(t \in d : d \in D)$$

- DF가 높으면 **널리 쓰이는 단어**(중요도 낮을 수 있음), 반대로 DF가 낮으면 특정 문맥에서만 쓰이는 **전문/특정 단어**(중요도 높을 수 있음).

## 역문서 빈도 (IDF)

- **정의:** 전체 문서 수를 문서 빈도로 나눈 값에 **로그를 취한 값**.

문서 집합에서 **드물게 등장할수록**(DF 낮을수록) 더 높은 가중치로 **중요도**를 반영.

- 문서 빈도가 높을수록 해당 단어가 일반적이므로 상대적으로 **중요하지 않**다는 의미.
- 문서 빈도의 **역수**를 취하면 드문 단어일수록 **IDF 값이 커짐** → 중요도 보정 역할.

- **수식 6.7 (역문서 빈도)**

$$IDF(t, D) = \log \left( \frac{\text{count}(D)}{1 + DF(t, D)} \right)$$

- 분모의 DF 값에 **1을 더함**: 특정 단어가 한 번도 등장하지 않을 때(분모=0) **0으로 나누는 문제 방지**.
- log 사용: 전체 문서 수를 분모로 나눈 값이 **너무 큰 값**이 되는 것을 방지하고 **정교한 가중치**를 얻기 위함.

## TF-IDF 정의

- TF-IDF는 **단어 빈도(TF)**와 **역문서 빈도(IDF)**를 곱한 값.

- **수식 6.8 (TF-IDF)**

$$TF-IDF(t, d, D) = TF(t, d) \times IDF(t, d)$$

- 문서 내에서 자주 등장하지만 전체 문서 내에 **적게 등장**하면 TF-IDF 값이 커진다.  
반대로 전체 문서에서 자주 등장하는 **관사·관용어** 등의 가중치는 낮아진다.

## Scikit-learn 라이브러리 설치

- `pip install scikit-learn`

(허깅페이스 트랜스포머를 설치했다면 자동으로 함께 설치된 경우도 있으나, ImportError가 나면 위 명령으로 설치)

## TF-IDF 클래스

```
tfidf_vectorizer = sklearn.feature_extraction.text.TfidfVectorizer(  
    input="content",  
    encoding="utf-8",  
    lowercase=True,  
    stop_words=None,  
    ngram_range=(1, 1),  
    max_df=1.0,  
    min_df=1,  
    vocabulary=None,  
    smooth_idf=True,  
)
```

- **입력값(input):** 입력 데이터의 형태. 기본값 `"content"` 는 문자열 데이터/바이트 입력을 의미. 파일 경로를 사용할 때는 `filename` .
- **인코딩(encoding):** 바이트/파일 입력 시 사용할 텍스트 인코딩 값.
- **소문자 변환(lowercase):** `True` 이면 모든 입력 텍스트를 소문자로 변환.
- **불용어(stop\_words):** 분석에 도움이 되지 않는 단어 목록. 지정 시 해당 단어들은 사전에 추가되지 않음.
- **N-gram 범위(ngram\_range):** 사용할 N-gram의 **최소·최대**. 예: `(1, 2)` 는 유니그램 + 바이그램 사용.
- **최댓값 문서 빈도(max\_df):** 전체 문서 중 **일정 횟수 이상** 등장한 단어를 불용어로 처리. 정수/실수 모두 가능(실수는 비율).
- **최솟값 문서 빈도(min\_df):** 전체 문서 중 **일정 횟수 미만** 등장한 단어를 불용어로 처리 (정수/비율).
- **단어사전(vocabulary):** 미리 구축된 **사전**이 있다면 사용(미입력 시 학습 시 자동 구성).

- **IDF 분모치(smooth\_idf):** IDFIDFIDF 계산 시 **분모에 1을 더함**(수식 6.7과 동일한 맥락).

## 예제 6.2 TF-IDF 계산

```
from sklearn.feature_extraction.text import TfidfVectorizer

corpus = [
    "That movie is famous movie",
    "I like that actor",
    "I don't like that actor"
]

tfidf_vectorizer = TfidfVectorizer()
tfidf_vectorizer.fit(corpus)
tfidf_matrix = tfidf_vectorizer.transform(corpus)

print(tfidf_matrix.toarray())
print(tfidf_vectorizer.vocabulary_)
```

- **fit:** 말뭉치로 **사전과 통계량을 학습**.
- **transform:** 학습된 사전·통계량으로 **입력을 변환**.
- **toarray:** TF-IDF 희소 행렬을 **(문서 수) × (단어 수) 배열**로 변환(행=문서, 열=단어).
- **vocabulary\_:** 사용된 **단어 사전**(키=단어, 값=색인).

TF-IDF에서 점수가 높은 순으로 상위 3개 색인을 정리하면 예시처럼 [[2, 3, 5], [0, 4, 6], [0, 1, 4]]가 되며,

사전과 매칭하면 각각 `['famous','is','movie']`, `['actor','like','that']`, `['actor','don','like']`.

- 빈도 기반 벡터화는 **문장 순서·문맥**을 고려하지 못하며, **문서 중요도**는 반영하지만 **단어 의미** 자체는 담고 있지 않다.

예: “나는 늘 바나나를 먹었다”, “그녀가 과일을 섭취한다”, “고양이는 우유를...”을 벡터화하면 **문장 간 유사도는 동일하게** 나올 수 있다.

## Word2Vec

- 2013년 구글 공개 임베딩 모델. **분포 가설(distributional hypothesis)** 기반: “같은 문맥에서 자주 함께 나타나는 단어들은 서로 유사한 의미를 가질 가능성이 높다.”
- **동시 발생(co-occurrence) 확률**을 이용해 단어 간 유사성을 측정.
- 예: “내일 자동차를 타고 부산에 간다” vs “내일 비행기를 타고 부산에 간다”에서 ‘자동차’/‘비행기’는 ‘타고/부산’ 등과 함께 나타나 유사 분포를 가짐.
- 이를 통해 **분산 표현(Distributed Representation)** 학습: 단어를 고차원 벡터 공간의 실수 벡터로 매핑하여 의미를 담는다.  
같은/유사한 문맥에 등장하는 단어는 벡터 공간에서 가까운 위치로 표현.
- 분산 표현은 빈도 기반 기법의 한계(의미 정보 미포착)를 극복하고, 문서 내 단어 관계와 벡터 공간상의 유사성을 반영.  
다양한 자연어 처리 작업에서 높은 성능을 보이며, 다운스트림 작업에도 효과적.

## 단어 벡터화

- 벡터화 방식: **희소 표현(sparse)** vs **밀집 표현(dense)**.
  - 원-핫, TF-IDF 등 빈도 기반 → **희소 표현**(대부분이 0).
  - Word2Vec 등 임베딩 → **밀집 표현**(실수 값이 꽉 찬 벡터).

표 6.3 단어의 희소 표현

소	0	1	0	0	0
일고	1	0	0	0	0
외양간	0	0	0	1	0
고친다	0	0	0	0	1

표 6.3 (단어의 희소 표현): ‘소/일고/외양간/고친다 ...’와 같은 항목에 대해 0/1로만 표현.

표 6.4 단어의 밀집 표현

소	0.3914	-0.1749	...	0.5912	0.1321
일고	-0.2893	0.3814	...	-0.1492	-0.2814
외양간	0.4812	0.1214	...	-0.2745	0.0132
고친다	-0.1314	-0.2809	...	0.2014	0.3016

표 6.4 (단어의 밀집 표현): 각 단어가 실수 값 벡터로 표현(예: 0.3914, -0.1749 ...).

- 희소 표현은 사전 크기가 커질수록 벡터도 커져 공간 낭비/연산 비용↑, 단어 간 유사성 반영 어려움.

밀집 표현은 고정 길이 실수 벡터로 효율적이며, 거리 계산 등 유사성 연산에 유리.

밀집 표현으로 학습한 단어 벡터를 단어 임베딩 벡터(Word Embedding Vector) 라고 하며, Word2Vec은 대표적인 임베딩 기법.

Word2Vec은 CBoW와 Skip-gram 두 방법을 사용.

## CBoW

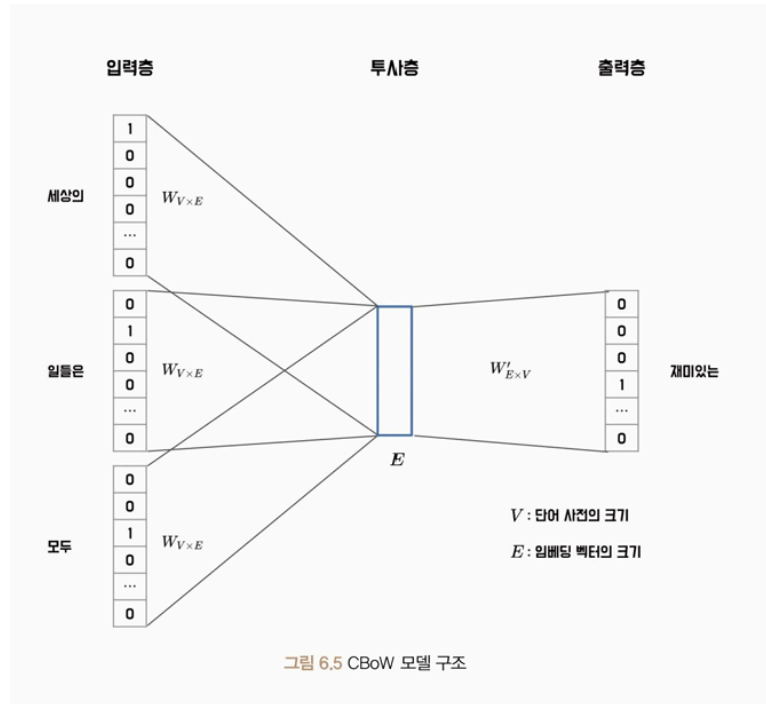
- CBoW(Continuous Bag of Words): 주변 단어(문맥)로 중심 단어를 예측.
  - 중심 단어(Center Word): 예측할 단어
  - 주변 단어(Context Word): 예측에 사용되는 단어들
  - 윈도우(Window): 중심 단어 기준으로 고려할 좌우 단어 개수
  - 슬라이딩 윈도우(Sliding Window): 윈도우를 이동시키며 여러 중심 단어-주변 단어 쌍을 생성

입력 문장	학습 데이터
(세상의 재미있는 일들은)모두 밤에 일어난다	(재미있는, 일들은   세상의)
(세상의 재미있는 일들은 모두)밤에 일어난다	(세상의, 일들은, 모두   재미있는)
(세상의 재미있는 일들은 모두 밤에)일어난다	(세상의, 재미있는, 모두, 밤에   일들은)
세상의(재미있는 일들은 모두 밤에 일어난다)	(재미있는, 일들은, 밤에, 일어난다   모두)
세상의 재미있는(일들은 모두 밤에 일어난다)	(일들은, 모두, 일어난다   밤에)
세상의 재미있는 일들은(모두 밤에 일어난다)	(모두, 밤에   일어난다)

그림 6.4 CBoW의 학습 데이터 구성

그림 6.4 (CBoW의 학습 데이터 구성): 문장 “세상의 재미있는 일들은 모두 밤에 일어난다”에서 윈도우=2일 때 (주변 단어, 중심 단어) 쌍 생성 예시.

- 그림 6.5 (CBoW 모델 구조):



입력층(원-핫) → 투사층(Projection, Lookup table/LUT) → 임베딩 평균 → 출력층(소프트맥스).

- 입력 원-핫 인덱스로 LUT에서 해당 임베딩 벡터를 조회
- 평균 임베딩을 구해 가중치 WWW 와 곱해 **V(어휘 크기)** 차원의 점수 벡터 산출
- 소프트맥스로 **중심 단어**를 예측

## Skip-gram

- 정의: CBOW와 반대로 **중심 단어를 입력**으로 받아 **주변 단어**를 예측.

입력 문장	학습 데이터
(세상의 재미있는 일들은)모두 밤에 일어난다	(세상의   재미있는), (세상의   일들은)
(세상의 재미있는 일들은 모두)밤에 일어난다	(재미있는   세상의), (재미있는   일들은), (재미있는   모두)
(세상의 재미있는 일들은 모두 밤에)일어난다	(일들은   세상의), (일들은   재미있는), (일들은   모두), (일들은   밤에)
세상의(재미있는 일들은 모두 밤에 일어난다)	(모두   재미있는), (모두   일들은), (모두   밤에), (모두   일어난다)
세상의 재미있는(일들은 모두 밤에 일어난다)	(밤에   일들은), (밤에   모두), (밤에   일어난다)
세상의 재미있는 일들은(모두 밤에 일어난다)	(일어난다   모두), (일어난다   밤에)

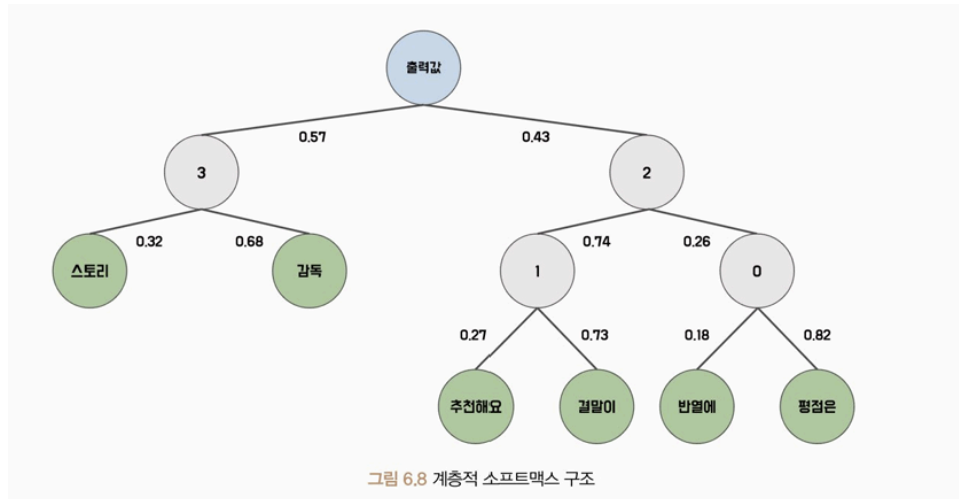
그림 6.6 Skip-gram의 학습 데이터 구성

그림 6.6 (Skip-gram의 학습 데이터 구성): 윈도우=2일 때 중심 단어 기준 양쪽 문맥으로 여러 쌍을 생성.

- 
- The diagram illustrates the Skip-gram model architecture. On the left, an input vector  $V$  (labeled '재미있는') is represented as a column vector  $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ \dots \\ 0 \end{bmatrix}$ . This vector is multiplied by the weight matrix  $W_{V \times E}$  to produce an embedding matrix  $E$ , shown as a blue-outlined rectangle. The embedding matrix  $E$  is then multiplied by three different weight matrices  $W'_{E \times V}$  to produce three output vectors. The top output vector, labeled '세상인', is  $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ \dots \\ 0 \end{bmatrix}$ . The middle output vector, labeled '일들은', is  $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ \dots \\ 0 \end{bmatrix}$ . The bottom output vector, labeled '모두', is  $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ \dots \\ 0 \end{bmatrix}$ . The diagram shows that the embedding matrix  $E$  is shared across all three output paths.
- $V$  : 단어 사전의 크기  
 $E$  : 임베딩 벡터의 크기
- 그림 6.7 Skip-gram 모델 구조

- 대규모 말뭉치/어휘에서 **소프트맥스 연산**은 **내적 연산**이 방대하여 학습 속도 저하.  
→ **계층적 소프트맥스**와 **네거티브 샘플링**으로 완화.

- **정의:** 출력층을 이진 트리(Binary tree) 로 구성하여 연산.
  - 자주 등장하는 단어일수록 트리의 상위 노드에, 드물게 등장하는 단어일수록 하위 노드에 배치.
  - 각 노드는 학습 가능한 벡터를 가지며, 입력값과 노드 벡터의 내적 후 시그모이드로 노드 통과 확률을 계산.
- **그림 6.8 (계층적 소프트맥스 구조):** 루트부터 앞 노드(단어) 까지의 경로 확률의 곱이 단어 확률.

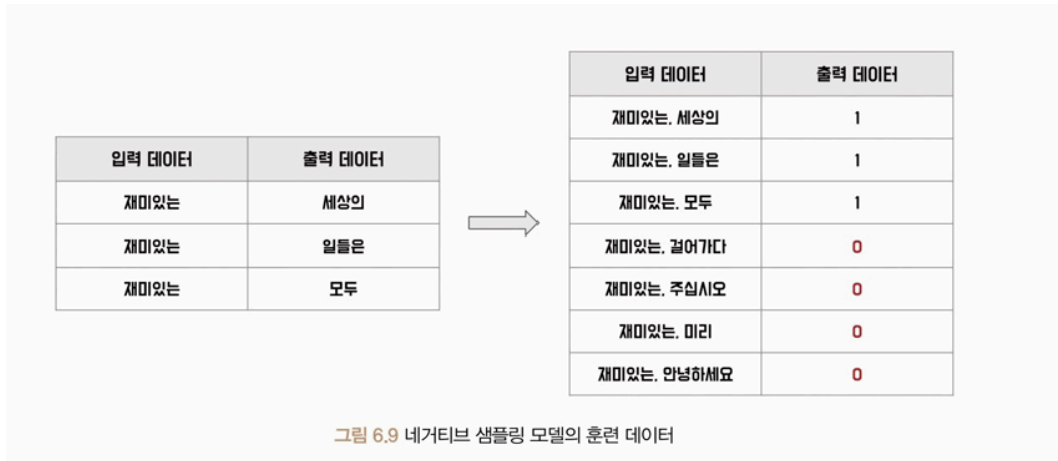


- 예: “추천해요”의 확률 =  $0.43 \times 0.74 \times 0.27 = 0.085914$ .
- 이 경우 **경로상 노드(예: 1, 2번)**의 벡터만 최적화하면 됨.
- 시간 복잡도
  - 일반적 소프트맥스:  $O(V)$
  - 계층적 소프트맥스:  $O(\log V)$

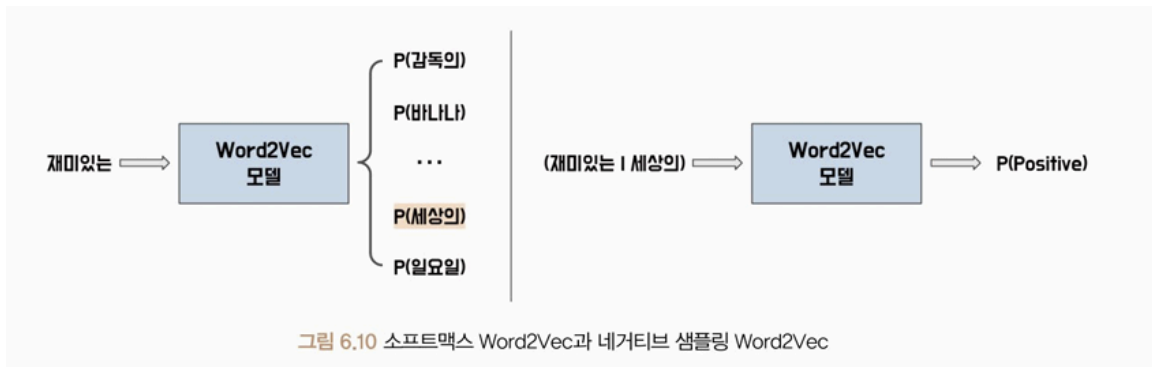
## 네거티브 샘플링

- **정의:** Word2Vec에서 사용하는 확률적 샘플링 기법. **전체 단어 집합**에서 일부 단어를 샘플링하여 **오답(negative)** 단어로 사용.
- **아이디어:** 학습 원도 내에 **등장하지 않는 단어를**  $n$ 개 추출해 **정답 단어**와 함께(정답=1, 오답=0) **소프트맥스 대신 이진 분류(로지스틱)** 연산으로 학습 → 전체 어휘에 대한 확률 계산 없이 **효율적 학습**.
- **샘플 수:** 일반적으로  $n \in [5, 20]$ .
- **수식 6.9 (네거티브 샘플링의 추출 확률)**

$$P(w_i) = \frac{f(w_i)^{0.75}}{\sum_j f(w_j)^{0.75}}$$
  - $f(w_i)$ : 말뭉치에서 단어  $w_i$ 의 **출현 빈도수**.
  - **0.75 제곱:** 너무 자주/너무 희귀한 단어의 치우침을 완화하는 **정규화 지수**(실험을 통해 얻어진 값).
  - 예) ‘추천해요’가 100번, 전체 단어가 2,000이면  $f = 100/2000 = 0.05$ .
- **학습 데이터 구성(그림 6.9):**



- 일반 Skip-gram의 (입력, 출력) 쌍에 더해, 동일 입력에 대해 샘플링된 오답 단어들을 레이블 0으로 추가.
- 즉, 이진 분류용 레이블(정답 1 / 오답 0)로 바뀜.
- 출력 구조 비교(그림 6.10):



- 소프트맥스 Word2Vec:  $P(\text{각 단어} \mid \text{입력})$  전체 분포를 산출.
- 네거티브 샘플링 Word2Vec: (입력, 출력 후보) 쌍에 대해  $P(\text{positive})$  (해당 쌍이 실제인지 여부)를 산출.
- 학습 시 레이블이 1이면 확률↑, 0이면 확률↓가 되도록 가중치 최적화.

## 모델 실습: Skip-gram

### 임베딩 클래스

```
embedding = torch.nn.Embedding(
    num_embeddings,    # 단어 사전 크기 V
    embedding_dim,    # 임베딩 차원 E
```

```
padding_idx=None, # 패딩 토큰 인덱스(있으면 해당 벡터는 0으로 고정)
max_norm=None,    # 임베딩 벡터의 최대 노름(넘으면 잘라서 제한)
norm_type=2.0     # 노름 종류(L2가 기본, 1로 설정 시 L1)
)
```

- **num\_embeddings**: 단어 사전 크기 V.
- **embedding\_dim**: 임베딩 차원 E.
- **padding\_idx**: 패딩용 토큰 인덱스(해당 벡터는 학습에서 최적화되지 않음).
- **max\_norm / norm\_type**: 임베딩 벡터 크기 제한(정규화).

### 예제 6.3 기본 Skip-gram 클래스

```
from torch import nn

class VanillaSkipgram(nn.Module):
    def __init__(self, vocab_size, embedding_dim):
        super().__init__()
        self.embedding = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embedding_dim
        )
        self.linear = nn.Linear(
            in_features=embedding_dim,
            out_features=vocab_size
        )
    def forward(self, input_ids):
        embeddings = self.embedding(input_ids)
        output = self.linear(embeddings)
        return output
```

- **구성**: 임베딩 조회(lookup) → 선형 변환 → (기본형에서는) 소프트맥스/크로스엔트로피로 학습.
- (본 장 실습 코드는 **계층적 소프트맥스·네거티브 샘플링 없이** 동작하는 기본형을 예시로 제시함.)

## 예제 6.4 영화 리뷰 데이터셋 전처리

```
import pandas as pd
from Korpora import Korpora
from konlpy.tag import Okt

corpus = Korpora.load("nsmc")
corpus = pd.DataFrame(corpus.test)

tokenizer = Okt()
tokens = [tokenizer.morphs(review) for review in corpus.text]
print(tokens[:3])
```

- **데이터**: NSMC 테스트셋을 불러와 **형태소** 단위로 토큰화.

## 예제 6.5 단어 사전 구축

```
from collections import Counter

def build_vocab(corpus, n_vocab, special_tokens):
    counter = Counter()
    for tokens in corpus:
        counter.update(tokens)
    vocab = special_tokens
    for token, count in counter.most_common(n_vocab):
        vocab.append(token)
    return vocab

vocab = build_vocab(tokens, n_vocab=5000, special_tokens=["<unk>"])
token_to_id = {token: idx for idx, token in enumerate(vocab)}
id_to_token = {idx: token for idx, token in enumerate(vocab)}

print(vocab[:10])
print(len(vocab))
```

- **n\_vocab**: 최대 어휘 수(예: 5,000).

- **special\_tokens:** "<unk>" 등 특수 토큰 포함.
- 결과: 사전 크기 5,001(특수 토큰 1개 + 상위 5,000개).

## 예제 6.6 Skip-gram의 단어 쌍 추출

```
def get_word_pairs(tokens, window_size):
    pairs = []
    for sentence in tokens:
        sentence_length = len(sentence)
        for idx, center_word in enumerate(sentence):
            window_start = max(0, idx - window_size)
            window_end = min(sentence_length, idx + window_size + 1)
            center_word = sentence[idx]
            context_words = sentence[window_start:idx] + sentence[idx+1:window_end]
            for context_word in context_words:
                pairs.append([center_word, context_word])
    return pairs

word_pairs = get_word_pairs(tokens, window_size=2)
print(word_pairs[:5])
```

- **window\_size:** 중심 단어 기준으로 좌우 몇 개를 고려할지.
- 출력: [중심, 주변] 쌍 목록.

## 예제 6.7 인덱스 쌍 변환

```
def get_index_pairs(word_pairs, token_to_id):
    pairs = []
    unk_index = token_to_id["<unk>"]
    for center_word, context_word in word_pairs:
        center_index = token_to_id.get(center_word, unk_index)
        context_index = token_to_id.get(context_word, unk_index)
        pairs.append([center_index, context_index])
    return pairs

index_pairs = get_index_pairs(word_pairs, token_to_id)
```

```
print(index_pairs[:5])
```

- 사전에 없으면 `<unk>` 인덱스로 대체.

## 예제 6.8 데이터로더 적용

```
import torch
from torch.utils.data import TensorDataset, DataLoader

index_pairs = torch.tensor(index_pairs)
center_indices = index_pairs[:, 0]
context_indices = index_pairs[:, 1]

dataset = TensorDataset(center_indices, context_indices)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)
```

- 인덱스 쌍을 **텐서**로 변환하여 **Dataset/DataLoader** 구성.

## 예제 6.9 Skip-gram 모델 준비 작업

```
from torch import optim
device = "cuda" if torch.cuda.is_available() else "cpu"

word2vec = VanillaSkipgram(vocab_size=len(token_to_id), embedding_dim
=128).to(device)
criterion = nn.CrossEntropyLoss().to(device)
optimizer = optim.SGD(word2vec.parameters(), lr=0.1)
```

- 임베딩 차원 예: 128.
- 손실: 분류 문제이므로 **크로스엔트로피**.

## 예제 6.10 모델 학습

```
for epoch in range(10):
    cost = 0.0
```

```

for input_ids, target_ids in dataloader:
    input_ids = input_ids.to(device)
    target_ids = target_ids.to(device)

    logits = word2vec(input_ids)
    loss = criterion(logits, target_ids)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    cost += loss

cost = cost / len(dataloader)
print(f"Epoch : {epoch+1:4d}, Cost : {cost:.3f}")

```

- 출력 예: Epoch : 1, Cost : 6.177 ... Epoch : 10, Cost : 5.791 .

## 예제 6.11 임베딩 값 추출 & 코사인 유사도

```

token_to_embedding = dict()
embedding_matrix = word2vec.embedding.weight.detach().cpu().numpy()
for word, embedding in zip(vocab, embedding_matrix):
    token_to_embedding[word] = embedding

index = 30
token = vocab[index]
token_embedding = token_to_embedding[token]
print(token)
print(token_embedding)

```

- 임베딩 행렬에서 단어↔벡터 매핑을 구성해 임베딩을 확인.

### 코사인 유사도(수식 6.10)

두 벡터  $a, b$ 의 각도 기반 유사도:

$$\text{cosine similarity}(a, b) = \frac{a \cdot b}{\|a\| \|b\|}$$

- 값이 **1에 가까울수록** 유사, **0에 가까울수록** 비유사.

```
E = model.embedding.weight.detach().cpu().numpy() # (V,E)
# 코사인으로 top-k 이웃 찾기 구현 (과제: np.dot 후 정규화)
```

### 한국어 전처리 팁

- 조사/어미가 많은 언어 → **형태소 분석(Okt, Mecab 등) 후 토큰화** 권장.
- 숫자/특수기호는 정규화. 하이픈/이모지 처리는 말뭉치 성격에 맞춤.

## 5) Gensim Word2Vec

```
from gensim.models import Word2Vec

w2v = Word2Vec(
    sentences=tokens, # 토큰 리스트의 리스트
    vector_size=128, # 임베딩 차원
    window=5, # 윈도우 크기
    min_count=1, # 최소 등장 횟수
    sg=1, # 1=Skip-gram, 0=CBOW
    hs=0, # 1=Hierarchical Softmax
    negative=5, # 네거티브 샘플 수(>0이면 NS 사용)
    workers=3, epochs=3, ns_exponent=0.75
)
# 유사어
w2v.wv.most_similar("연기", topn=5)
# 코사인
w2v.wv.similarity("연기", "연기력")
# 저장/로드
w2v.save("w2v.model")
```

### 매개변수 한 줄 정리

- **sg** : 1=Skip-gram(희귀 단어 강점), 0=CBOW(빠름).
- **negative** vs **hs** : 보통 **negative>0, hs=0** 권장.
- **min\_count** : 노이즈 제거. 작은 코퍼스면 1~2, 크면 5+.
- **window** : 문맥 폭. 일반 3~10.

- `vector_size` : 100~300 권장(작업 규모에 맞춤).

## 6) fastText: 서브워드(하위 단어) 임베딩

### 왜 필요한가?

- 한국어처럼 접사/활용이 풍부하거나 **OOV**(사전에 없는 단어) 처리에 강함.
- 단어를 문자 N-gram으로 분해해 **단어 벡터 = (서브워드 벡터 합/평균)**.

### 핵심 파라미터

- `min_n` , `max_n` : 문자 n-gram 범위(예: 3~6).
- `sg` , `window` , `vector_size` , `negative` , `epochs` 등은 Word2Vec과 동일 개념.

### Gensim fastText 예시

```
from gensim.models import FastText
ft = FastText(
    sentences=tokens, vector_size=128, window=5, min_count=5,
    sg=1, negative=5, epochs=3, min_n=3, max_n=6
)
ft.wv.most_similar("서울특별시") # 서브워드 덕분에 희귀/변형에도 강함
```

### 장점

- 형태 변화/신조어/OOV에 견고.

### 주의

- 학습·메모리 비용 ↑, 하이퍼파라미터(특히 n-gram 범위) 영향 큼.

## 7) 계층적 소프트맥스 vs 네거티브 샘플링 선택

- 데이터 작을데 **V**가 크다: HS 유리(안정,  $O(\log V)O(\log V)O(\log V)$ ).
- 데이터/코퍼스가 크다: Negative Sampling 일반적(속도·성능 균형).
- 희귀 단어 성능: Skip-gram + NS가 보편적 베이스라인.

## 8) 평가 & 활용

### 내재적 평가(빠른 점검)

- 이웃 단어 품질: `most_similar` , 코사인 유사도 분포 확인.

- 유사도 태스크(WordSim, Kor-STs 등) 있으면 상관계수로 점수화.

## 외재적 평가(권장)

- 다운스트림(분류·개체명 인식 등) 성능 비교.

## 시각화

- 임베딩을 UMAP/T-SNE로 2D 투영 → 군집/주제 확인.

## 9) 정리

- **토큰라이저**: Mecab(Okt보다 견고), 띄어쓰기 보정 여부.
- **불용어 사전**: 조사/어미/기호 최소화(단, fastText는 과도 필터링 지양).
- **정규화**: 숫자/반복자·이모지 처리 규칙 결정.
- **단어 사전 크기**: `min_count` 로 제어.
- **윈도우**: 문체/도메인에 맞게(뉴스 5±, 짧은 SNS 3±).
- **seed/재현성**: 난수 고정, 저장은 `.wv` (KeyedVectors)도 병행.

## 10) FAQ

- **IDF에 +1을 더하는 이유?** 분모 0 방지·안정화(smoothing).
- **TF-IDF 한계?** 순서·문맥 미반영, 희소, 의미 정보 빈약.
- **Skip-gram 장점?** 희귀 단어·표현에 강함(컨텍스트 다양 채집).
- **Negative Sampling 분포 지수 0.75 이유?** 너무 흔한 단어 과표집 억제 & 학습 안정 (경험적 최적).
- **fastText가 OOV에 강한 이유?** 서브워드 합성으로 미등록 단어도 부분 정보로 벡터 구성.

## 11) 미니 코드 스니펫 모음

### 코사인 유사도 행렬 & Top-k

```
import numpy as np
from numpy.linalg import norm

def cosine_matrix(a, B): # a: (E,), B: (V,E)
    return (B @ a) / (norm(a) * norm(B, axis=1) + 1e-9)
```

```
vec = w2v.wv["연기"]          # (E,)
M  = w2v.wv.vectors          # (V,E)
cs = cosine_matrix(vec, M)
top = cs.argsort()[-6:][::-1] # 자기 자신 포함 주의
[w2v.wv.index_to_key[i] for i in top]
```

## 토큰→인덱스 매핑

```
vocab = {tok:i for i,tok in enumerate(my_tokens)}
id2tok = {i:t for t,i in vocab.items()}
```

## 12) 선택 가이드

- 빠른 베이스라인: `Word2Vec(sg=1, negative=5, vector_size=128, window=5, min_count=2)`
- 형태 변화·OOV 많음: `fastText(min_n=3, max_n=6)`
- 코퍼스 매우 작음: CBOW 시도(속도↑), HS 고려
- 과제/리포트 시각화: TF-IDF 중요단어 Top-k + 임베딩 군집 2D 투영 함께 제시

## 예제 6.17 fastText OOV 처리

- Word2Vec은 사전에 없는 OOV 단어의 임베딩을 계산하지 못하지만, **fastText**는 하위 단어(subword) 단위로 분해해 OOV의 임베딩을 계산할 수 있다.
- `fastText.wv.index_to_key` 는 학습된 **단어 사전 리스트**. `'사랑해요'` 는 리스트에 없으므로 OOV로 처리된다.
- fastText는 `'사랑해요'` 를 `'사랑'`, `'랑해'`, `'해요'` 등의 하위 단어로 분해하고, 이들의 임베딩을 조합해 전체 토큰 임베딩을 만든다. 그래서 다른 문맥에서 등장했던 하위 단어의 정보로 OOV 토큰 임베딩을 유추할 수 있다.

## 코드

```
oov_token = "사랑해요"
oov_vector = fastText.wv[oov_token]

print(oov_token in fastText.wv.index_to_key)
```

```
print(fastText.wv.most_similar(oov_vector, topn=5))
```

## 출력

- False
- [('사랑', ...), ('사랑에', ...), ('사랑의', ...), ('사랑을', ...), ('사랑하는', ...)]  
→ OOV인 '사랑해요'의 벡터가 의미적으로 가까운 단어들과 높은 유사도를 보임.

## 순환 신경망

### 연속적인 데이터와 의존성

- RNN(Recurrent Neural Network)은 순서가 있는 연속 데이터(Sequence data)에 적합. 각 시점(time step)의 데이터가 이전 시점의 영향을 받는 특성에 대응한다.
- 예시(자연어): “금요일이 지나면, ... / 수요일이 지나면 목요일이고, 목요일이 지나면 금요일, 금요일이 지나면, ...”
  - 이전 단어(또는 앞 문장)의 패턴/의미가 뒤 단어 선택에 영향을 준다.
  - 문장 전체에 걸친 상관관계(correlation)를 고려해야 하므로, 자연어는 연속형 데이터의 성격을 가진다.

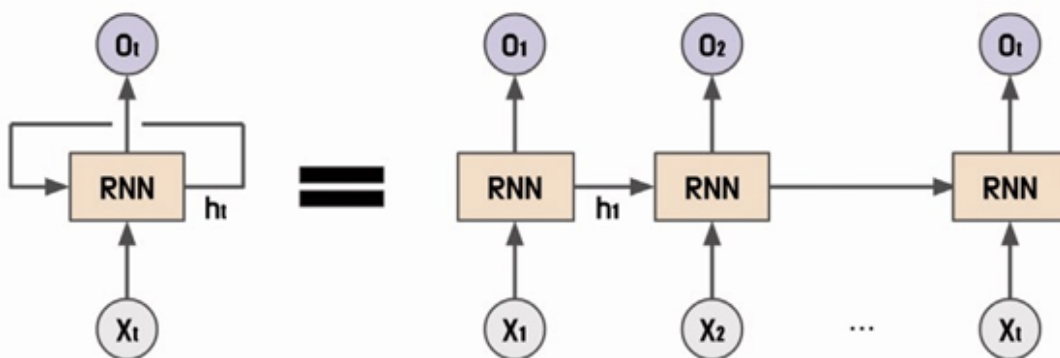


그림 6.12 순환 신경망의 셀

- 매 시점  $t$ : 입력  $x_t$ 와 직전 은닉 상태  $h_{t-1}$ 로 현재 은닉 상태  $h_t$  및 출력  $y_t$ 를 계산한다.
- 노드(셀)는 시점마다 은닉 상태(hidden state)를 유지한다.

### 수식 6.11 순환 신경망의 은닉 상태

$$h_t = \sigma_h(h_{t-1}, x_t)$$

$$h_t = \sigma_h(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

- $\sigma_h$ : 은닉 상태 계산용 **활성화 함수**
- $W_{hh}$ : 이전 은닉 상태  $h_{t-1}$ 에 대한 가중치
- $W_{xh}$ : 입력값  $x_t$ 에 대한 가중치
- $b_h$ : 은닉 상태 편향

## 수식 6.12 순환 신경망의 출력값

$$y_t = \sigma_y(h_t)$$

$$y_t = \sigma_y(W_{hy}h_t + b_y)$$

- $\sigma_y$ : 출력 계산용 **활성화 함수**
- $W_{hy}$ : 현재 은닉 상태  $h_t$ 에 대한 가중치,  $b_y$ : 출력 편향

## 일대다 구조 (One-to-Many)

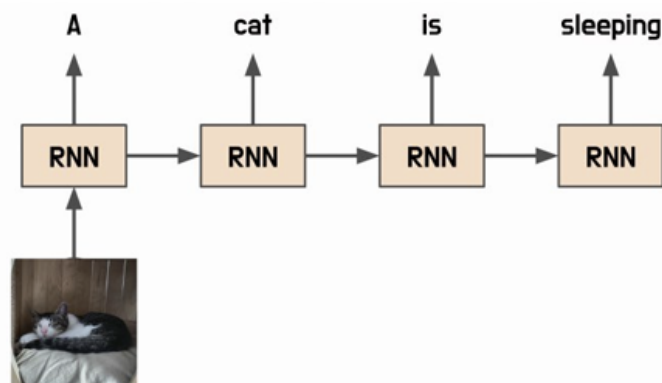


그림 6.13 순환 신경망의 일대다 구조

- 하나의 **입력 시퀀스** → 여러 개의 **출력값**.
- 자연어 예: 문장을 입력받아 **각 단어의 품사**를 예측(문장 → 단어 단위 출력).
- 이미지 캡셔닝처럼 **이미지 한 장** → **설명 시퀀스**도 일대다.

## 다대일 구조 (Many-to-One)

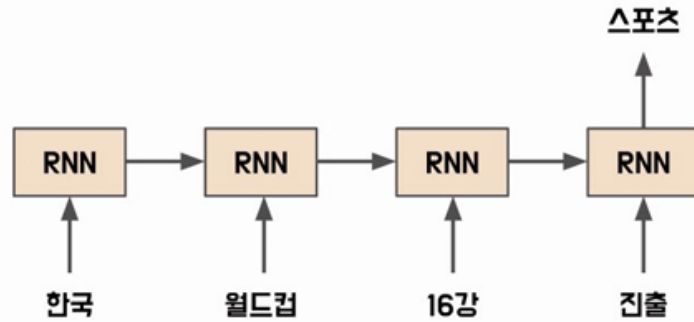


그림 6.14 순환 신경망의 다대일 구조

- 여러 입력 → 하나의 출력.
- 예: 감성 분류(문장 전체 입력 → 긍/부정), 문장 분류, 문장 간 관계를 추론하는 자연어 추론(NLI) 등.

## 다대다 구조 (Many-to-Many) (그림 6.15)

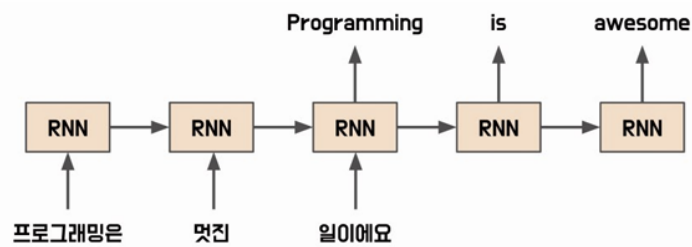


그림 6.15 순환 신경망의 다대다 구조

- 입력 시퀀스와 출력 시퀀스가 둘 다 여러 시점.
- 예: 번역(입력 문장 → 출력 문장), 음성 인식(음성 → 문자).
- 입력/출력 길이가 다를 수 있어 **시퀀스-시퀀스(Seq2Seq)** 구조로 구현: **인코더**(입력 처리) + **디코더**(출력 생성).

## 양방향 순환 신경망 (BiRNN)

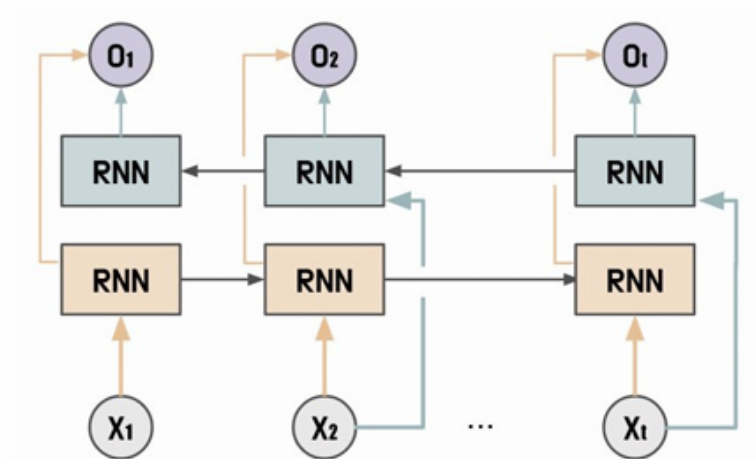


그림 6.16 양방향 순환 신경망 구조

- 기본 RNN이  $t-1$ 의 정보만 쓰는 데 비해, BiRNN은 **이전( $t-1$ )과 이후( $t+1$ ) 방향 정보**를 함께 고려해 현재  $t$ 의 출력을 계산.
- 자연어처럼 **앞·뒤 문맥**이 모두 중요한 데이터에서 유리하다.

## 다중 순환 신경망 (Stacked RNN)

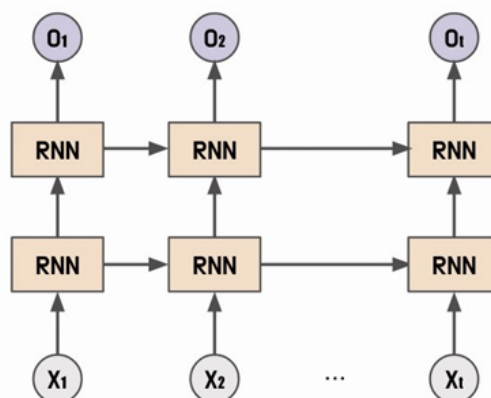


그림 6.17 다중 순환 신경망 구조

- 여러 개의 RNN을 **층으로 쌓은 구조**. 각 층의 출력이 다음 층 입력으로 전달되어 **더 복잡한 패턴**을 학습.
- 장점: 다양한 특징 추출, 성능 향상 가능.
- 주의: **층이 깊어질수록 학습 시간 증가, 기울기 소실 가능성↑**.

## 순환 신경망 클래스 (PyTorch)

## 클래스 시그니처(원문)

```
rnn = torch.nn.RNN(
    input_size,
    hidden_size,
    num_layers=1,
    nonlinearity="tanh",
    bias=False,
    batch_first=True,
    dropout=0,
    bidirectional=False
)
```

## 하이퍼파라미터(원문 설명 요약)

- **input\_size**: 입력 특성 크기(=각 시점의 입력 차원).
- **hidden\_size**: 은닉 상태 크기.
- **num\_layers**: RNN 층 수( $\geq 2$ 면 다중 RNN).
- **nonlinearity**: 은닉 활성화 함수 종류( `tanh` , `relu` ).
- **bias**: 편향 사용 여부.
- **batch\_first**: 입력/출력의 첫 차원을 배치로 둘지 여부.
- **dropout**: 드롭아웃 확률(과대적합 방지).
- **bidirectional**: 양방향 RNN 여부.

## 예제 6.18 양방향 다층 신경망

### 코드 (원문)

```
import torch
from torch import nn

input_size = 128
output_size = 256
num_layers = 3
bidirectional = True
```

```
model = nn.RNN(
    input_size=input_size,
    hidden_size=output_size,
    num_layers=num_layers,
    nonlinearity="tanh",
    batch_first=True,
    bidirectional=bidirectional,
)
```

```
batch_size = 4
```

```
sequence_len = 6
```

```
inputs = torch.randn(batch_size, sequence_len, input_size)
h_0 = torch.rand(num_layers * (int(bidirectional) + 1), batch_size, output_size)
```

```
outputs, hidden = model(inputs, h_0)
print(outputs.shape)
print(hidden.shape)
```

### 출력 (원문)

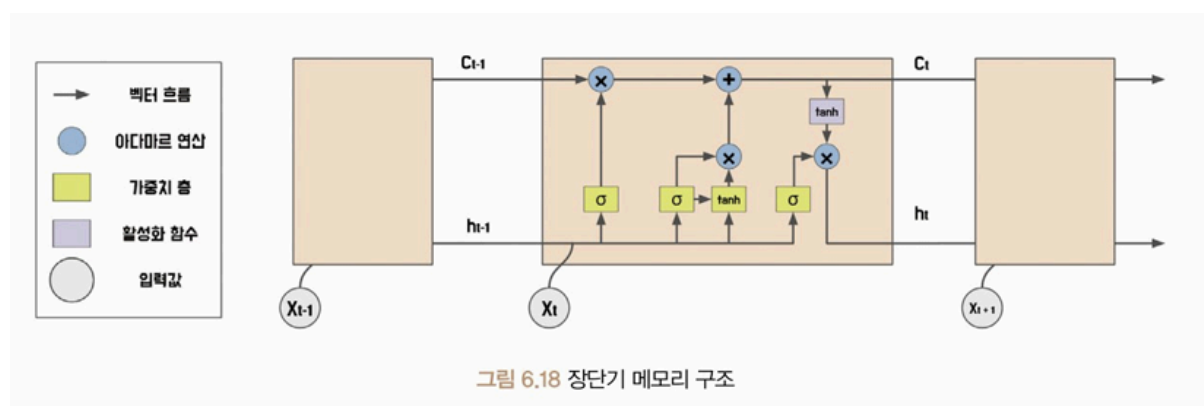
```
torch.Size([4, 6, 512])
torch.Size([6, 4, 256])
```

### 동작/모양 정리

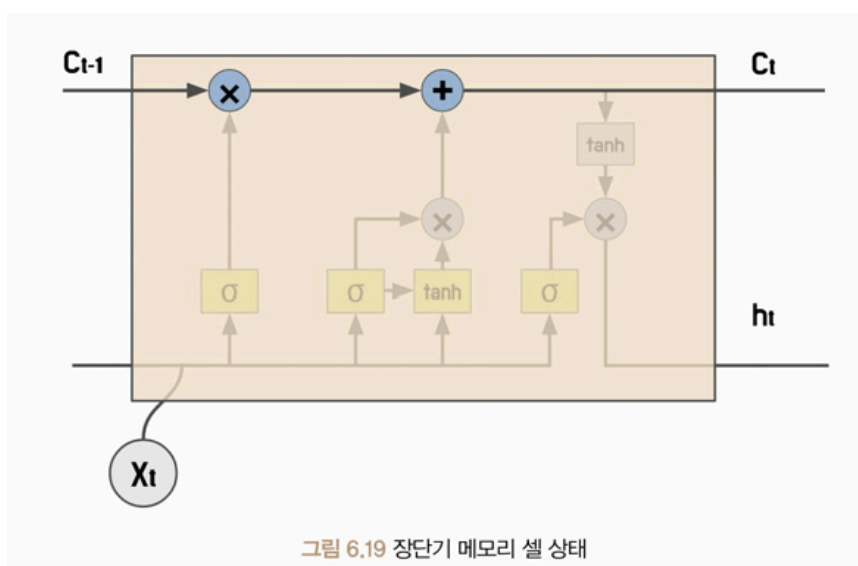
- 입력 `inputs` 크기: **[배치, 시퀀스 길이, 입력 특성]** (배치 우선).
- 초기 은닉 `h_0` 크기: **[ $(\text{계층 수} \times (\text{양방향 여부} + 1))$ , 배치, 은닉 크기]**.
- 순방향 연산 결과 `outputs` 와 최종 은닉 `hidden` 을 반환:
  - `outputs` : **[배치, 시퀀스 길이,  $(\text{양방향 여부} + 1) \times \text{은닉 크기}$ ]**
  - `hidden` : 초기 은닉과 동일 차원으로 반환.

## 장단기 메모리 (LSTM)

- 일반 RNN은 이전 정보가 멀리 떨어져 있으면 잘 기억하지 못해 **장기 의존성 문제**와 **기울기 소실/폭주**가 발생할 수 있다.
- **장단기 메모리(LSTM)** 는 **메모리 셀(memory cell)** 과 **게이트(gate)** 를 도입해 정보를 **지우고(망각)**, **추가하고(기억)**, **내보내는(출력)** 흐름을 제어하여 위 문제를 완화한다.
- LSTM 전체 구조: 셀 상태  $C_t$  흐름과 세 게이트(망각/기억/출력)로 구성. 셀 상태는 정보를 저장·유지하며, 각 게이트가 이를 조절한다.



- 셀 상태: 현재 입력  $x_t$ , 이전 은닉  $h_{t-1}$  를 바탕으로 정보를 계산·저장하되, 셀 자체는 직접 출력값 계산에 쓰이지 않는다(게이트를 통해 최종 출력에 반영).



### 수식 6.13 망각 게이트

$$f_t = \sigma(W_x^{(f)}x_t + W_h^{(f)}h_{t-1} + b^{(f)})$$

- $f_t \in [0, 1]$ : 1에 가까울수록 이전 셀 상태를 더 유지, 0에 가까울수록 더 많이 삭제.

#### 수식 6.14 기억 게이트

$$g_t = \tanh\left(W_x^{(g)}x_t + W_h^{(g)}h_{t-1} + b^{(g)}\right), \quad i_t = \sigma\left(W_x^{(i)}x_t + W_h^{(i)}h_{t-1} + b^{(i)}\right)$$

- $g_t \in [-1, 1]$ : 새로 추가할 후보 정보,  $i_t \in [0, 1]$ : 후보 정보를 얼마나 반영할지 결정.

#### 수식 6.15 메모리 셀

$$c_t = f_t \odot c_{t-1} + g_t \odot i_t$$

- $\odot$ : 아다마르 곱(원소별 곱). 망각·기억 정보를 결합해 현재 셀 상태를 계산.

#### 수식 6.16 출력 게이트

$$o_t = \sigma\left(W_x^{(o)}x_t + W_h^{(o)}h_{t-1} + b^{(o)}\right)$$

#### 수식 6.17 은닉 상태 갱신

$$h_t = o_t \odot \tanh(c_t)$$

- 출력 게이트와  $\tanh(c_t)$  를 결합해 현재 은닉 상태를 결정.

#### 장단기 메모리 클래스 (PyTorch)

```
lstm = torch.nn.LSTM(
    input_size,
    hidden_size,
    num_layers=1,
    bias=False,
    batch_first=True,
    dropout=0,
    bidirectional=False,
    proj_size=0
)
```

- LSTM은 활성화 함수를 별도 지정하지 않으며, **투사 크기(proj\_size)** 로 계층 출력의 **선형 투사(Linear Projection)** 차원을 정함(그림 주석의 각주 설명).

#### 예제 6.19 양방향 다층 장단기 메모리

```

import torch
from torch import nn

input_size = 128
ouput_size = 256
num_layers = 3
bidirectional = True
proj_size = 64

model = nn.LSTM(
    input_size=input_size,
    hidden_size=ouput_size,
    num_layers=num_layers,
    batch_first=True,
    bidirectional=bidirectional,
    proj_size=proj_size,
)

batch_size = 4
sequence_len = 6

inputs = torch.randn(batch_size, sequence_len, input_size)
h_0 = torch.rand(
    num_layers * (int(bidirectional) + 1), batch_size,
    proj_size if proj_size > 0 else ouput_size,
)
c_0 = torch.rand(
    num_layers * (int(bidirectional) + 1), batch_size, ouput_size
)

outputs, (h_n, c_n) = model(inputs, (h_0, c_0))
print(outputs.shape)
print(h_n.shape)
print(c_n.shape)

```

## 출력

```
torch.Size([4, 6, 128])
torch.Size([6, 4, 64])
torch.Size([6, 4, 256])
```

## 텐서 크기 정리

- 입력 `inputs` : [배치, 시퀀스 길이, 입력 특성]
- 초기 은닉 `h_0` : [(계층 수 × (양방향 여부+1)), 배치, (proj\_size>0이면 proj\_size, 아니면 hidden\_size)]
- 초기 셀 `c_0` : [(계층 수 × (양방향 여부+1)), 배치, hidden\_size]
- 출력 `outputs` : [배치, 시퀀스 길이, (proj\_size>0이면 proj\_size, 아니면 hidden\_size)]
- 최종 은닉 `h_n` : 초기 은닉과 동일 차원, 최종 셀 `c_n` : 초기 셀과 동일 차원.

## 용어·하이퍼파라미터

- 셀 상태 `c_t`: 정보를 저장·유지하는 메모리 경로.
- 망각 게이트 `f_t`: 이전 정보를 얼마나 유지/삭제할지 결정.
- 기억 게이트 `g_t, i_t`: 새 정보 후보와 그 반영 비율 결정.
- 출력 게이트 `o_t`: 현재 은닉 상태에 반영할 정도 결정.
- `input_size` : 각 시점 입력 특성 크기.
- `hidden_size` : 셀 상태/은닉 상태의 기본 차원.
- `num_layers` : 층 수(≥2면 다층).
- `batch_first` : (배치, 시퀀스, 특성) 순서 사용 여부.
- `bidirectional` : 양방향 LSTM 사용 여부.
- `proj_size` : 선형 투사 차원(0이면 투사 미사용).

## 모델 실습

- 본 절에서는 순환 신경망과 LSTM 을 활용해 문장 긍정/부정 분류 모델을 학습한다.
- Input Review(텍스트) → RNN, RNN, RNN → Dropout → classifier(출력은 `hidden_dm` 으로 표기).

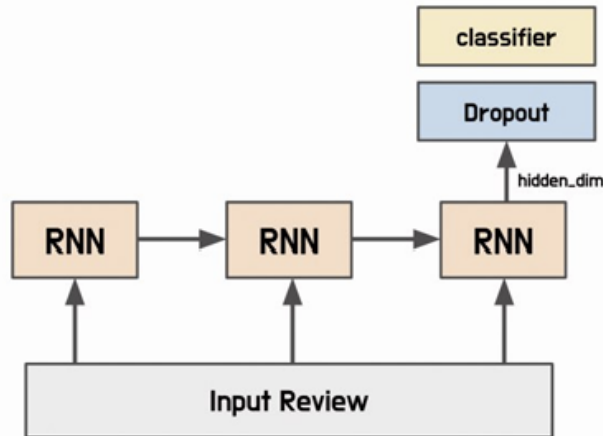


그림 6.23 긍정/부정 분류 모델 구조

- **임베딩 계층**: [어휘 사전의 크기] × [임베딩 벡터 크기] 형태로 순환 전 단계에 위치.
- 임베딩은 무작위 초기화 또는 사전 학습된 임베딩(6.4 Word2Vec 절) 을 적용하는 두 가지 방법을 모두 다룬다.

## 예제 6.20 문장 분류 모델

### 핵심 요약

- `SentenceClassifier` 는 임베딩 층 → (RNN/LSTM) → 드롭아웃 → 선형 분류기로 구성된다.
- `model_type` 으로 `rnn` 또는 `lstm` 을 선택하며, `bidirectional=True` 일 때 분류기의 입력 차원을 `hidden_dim*2` 로 잡는다.
- 순환 층의 출력 시퀀스 중 마지막 시점( `output[:, -1, :]` )만 분류기로 전달하여 이진 로짓을 반환한다.

### 코드

```

from torch import nn

class SentenceClassifier(nn.Module):
    def __init__(
        self,
        n_vocab,
        hidden_dim,
        embedding_dim,
        n_layers,
    ):

```

```

        dropout=0.5,
        bidirectional=True,
        model_type="lstm"
    ):
        super().__init__()

        self.embedding = nn.Embedding(
            num_embeddings=n_vocab,
            embedding_dim=embedding_dim,
            padding_idx=0
        )

        if model_type == "rnn":
            self.model = nn.RNN(
                input_size=embedding_dim,
                hidden_size=hidden_dim,
                num_layers=n_layers,
                bidirectional=bidirectional,
                dropout=dropout,
                batch_first=True,
            )
        elif model_type == "lstm":
            self.model = nn.LSTM(
                input_size=embedding_dim,
                hidden_size=hidden_dim,
                num_layers=n_layers,
                bidirectional=bidirectional,
                dropout=dropout,
                batch_first=True,
            )

        if bidirectional:
            self.classifier = nn.Linear(hidden_dim * 2, 1)
        else:
            self.classifier = nn.Linear(hidden_dim, 1)
        self.dropout = nn.Dropout(dropout)

    def forward(self, inputs):

```

```

embeddings = self.embedding(inputs)
output, _ = self.model(embeddings)
last_output = output[:, -1, :]
last_output = self.dropout(last_output)
logits = self.classifier(last_output)
return logits

```

## 용어·하이퍼파라미터

- `n_vocab` : 단어 사전 크기
- `embedding_dim` : 임베딩 벡터 차원
- `hidden_dim` : 순환 은닉 차원
- `n_layers` : 순환 계층 수( $\geq 2$ 면 다층)
- `bidirectional` : 양방향 여부
- `dropout` : 과적합 방지용 드롭아웃 확률
- 출력: 로짓(시그모이드 전)

## 예제 6.21 데이터세트 불러오기

- `Korpora` 의 네이버 영화 리뷰 감성 분석 데이터(`nsmc`) 를 로드한다.
- 제공된 `corpus.test` 를 기반으로 **90%/10%** 무작위 분할을 수행(출력 예: train 45,000 / test 5,000).

## 코드

```

import pandas as pd
from Korpora import Korpora

corpus = Korpora.load("nsmc")
corpus_df = pd.DataFrame(corpus.test)

train = corpus_df.sample(frac=0.9, random_state=42)
test = corpus_df.drop(train.index)

print(train.head(5).to_markdown())

```

```
print("Training Data Size :", len(train))
print("Testing Data Size :", len(test))
```

## 예제 6.22–6.23 토큰나이저, 단어 사전, 정수 인코딩/패딩

- `konlpy.tag.Okt` 로 토큰화 후 상위 5,000개 빈도 단어로 사전 구축.
- 특수 토큰으로 `<pad>`, `<unk>` 를 포함하여 총 5,002개 사전이 만들어진다(출력 예에서 첫 10개 토큰과 길이 확인).
- 각 문장을 정수 시퀀스로 변환하고, `max_length=32` 기준으로 자르기/패딩한다.

## 코드

```
from konlpy.tag import Okt
from collections import Counter
import numpy as np

def build_vocab(corpus, n_vocab, special_tokens):
    counter = Counter()
    for tokens in corpus:
        counter.update(tokens)
    vocab = special_tokens
    for token, count in counter.most_common(n_vocab):
        vocab.append(token)
    return vocab

tokenizer = Okt()
train_tokens = [tokenizer.morphs(review) for review in train.text]
test_tokens = [tokenizer.morphs(review) for review in test.text]

vocab = build_vocab(corpus=train_tokens, n_vocab=5000, special_tokens=
["<pad>", "<unk>"])
token_to_id = {token: idx for idx, token in enumerate(vocab)}
id_to_token = {idx: token for idx, token in enumerate(vocab)}

print(vocab[:10]); print(len(vocab))
```

```

import numpy as np

def pad_sequences(sequences, max_length, pad_value):
    result = []
    for sequence in sequences:
        sequence = sequence[:max_length]
        pad_length = max_length - len(sequence)
        padded_sequence = sequence + [pad_value] * pad_length
        result.append(padded_sequence)
    return np.asarray(result)

unk_id = token_to_id["<unk>"]
train_ids = [[token_to_id.get(token, unk_id) for token in review] for review in
train_tokens]
test_ids = [[token_to_id.get(token, unk_id) for token in review] for review in
test_tokens]

max_length = 32
pad_id = token_to_id["<pad>"]
train_ids = pad_sequences(train_ids, max_length, pad_id)
test_ids = pad_sequences(test_ids, max_length, pad_id)

print(train_ids[0]); print(test_ids[0])

```

- `pad_sequences` 는 긴 문장 자르기/짧은 문장 패딩을 모두 수행.
- 출력 예시에서 OOV 토큰은 1( "`<unk>`" )로 인코딩되고, 패딩으로 0( "`<pad>`" )가 뒤를 채운다.

## 예제 6.24 데이터로더 적용

- 정수 행렬과 레이블을 텐서로 변환해 `TensorDataset` → `DataLoader` 로 감싼다.

## 코드

```

import torch
from torch.utils.data import TensorDataset, DataLoader

```

```

train_ids = torch.tensor(train_ids)
test_ids = torch.tensor(test_ids)

train_labels = torch.tensor(train.label.values, dtype=torch.float32)
test_labels = torch.tensor(test.label.values, dtype=torch.float32)

train_dataset = TensorDataset(train_ids, train_labels)
test_dataset = TensorDataset(test_ids, test_labels)

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)

```

## 예제 6.25–6.26 손실/최적화, 학습·평가 루프

- **손실:** `BCEWithLogitsLoss` (시그모이드+BCELoss 결합형)
- **옵티마이저:** `RMSprop(lr=0.001)` — **지수 가중 이동 평균(EWMA)** 으로 기울기 규모를 추적해 학습률을 조절
- **평가:** `torch.sigmoid(logits) > 0.5` 로 정확도 계산, 일정 스텝 간격으로 **Train Loss**, 에포크 종료 시 **Val Loss/Accuracy** 출력

## 코드

```

from torch import optim, nn
import numpy as np
device = "cuda" if torch.cuda.is_available() else "cpu"

n_vocab = len(token_to_id)
hidden_dim = 64; embedding_dim = 128; n_layers = 2

classifier = SentenceClassifier(
    n_vocab=n_vocab, hidden_dim=hidden_dim,
    embedding_dim=embedding_dim, n_layers=n_layers
).to(device)

criterion = nn.BCEWithLogitsLoss().to(device)

```

```
optimizer = optim.RMSprop(classifier.parameters(), lr=0.001)
```

```
def train(model, datasets, criterion, optimizer, device, interval):
    model.train()
    losses = []
    for step, (input_ids, labels) in enumerate(datasets):
        input_ids = input_ids.to(device)
        labels = labels.to(device).unsqueeze(1)

        logits = model(input_ids)
        loss = criterion(logits, labels)
        losses.append(loss.item())

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    if step % interval == 0:
        print(f"Train Loss {step} : {np.mean(losses)}")

def test(model, datasets, criterion, device):
    model.eval()
    losses, corrects = [], []
    for step, (input_ids, labels) in enumerate(datasets):
        input_ids = input_ids.to(device)
        labels = labels.to(device).unsqueeze(1)
        logits = model(input_ids)
        loss = criterion(logits, labels)
        losses.append(loss.item())
        yhat = torch.sigmoid(logits) > .5
        corrects.extend(torch.eq(yhat, labels).cpu().tolist())
    print(f"Val Loss : {np.mean(losses)}, Val Accuracy : {np.mean(corrects)}")

epochs = 5
interval = 500
```

```
for epoch in range(epochs):
    train(classifier, train_loader, criterion, optimizer, device, interval)
    test(classifier, test_loader, criterion, device)
```

## 결과

- 초기: Val Loss  $\approx 0.4889$ , Val Accuracy  $\approx 0.7622$
- 후반 에포크: Val Loss  $\approx 0.3924$ , Val Accuracy  $\approx 0.8236$ 
  - $\Rightarrow$  검증 손실 감소, 정확도 상승을 확인.

## 예제 6.27 학습된 모델로부터 임베딩 추출

- 학습 완료 후 임베딩 층의 **가중치 행렬**에서 각 단어의 임베딩 벡터를 얻는다.
- 예시 출력: 특정 토큰(예: 사전에 있는 임의의 토큰)에 대한 실수 벡터가 출력된다.

## 코드

```
token_to_embedding = dict()
embedding_matrix = classifier.embedding.weight.detach().cpu().numpy()

for word, emb in zip(vocab, embedding_matrix):
    token_to_embedding[word] = emb

token = vocab[1000]
print(token, token_to_embedding[token])
```

## 예제 6.28 사전 학습 임베딩으로 임베딩 계층 초기화

- 6.4 절에서 학습한 **Word2Vec** 임베딩을 불러와, `<pad>`, `<unk>` 를 제외한 토큰의 초기 임베딩으로 설정한다.
- `nn.Embedding.from_pretrained` 로 임베딩 층을 생성한다.

## 코드

```

from gensim.models import Word2Vec
import numpy as np
from torch import nn
word2vec = Word2Vec.load("../models/word2vec.model")

init_embeddings = np.zeros((n_vocab, embedding_dim))
for index, token in id_to_token.items():
    if token not in ["<pad>", "<unk>"]:
        init_embeddings[index] = word2vec.wv[token]

embedding_layer = nn.Embedding.from_pretrained(
    torch.tensor(init_embeddings, dtype=torch.float32)
)

```

## 사전학습 임베딩 적용

```

class SentenceClassifier(nn.Module):
    def __init__(self, n_vocab, hidden_dim, embedding_dim, n_layers,
                  pretrained_embedding=None, bidirectional=True, dropout=0.5, model_type="lstm"):
        super().__init__()
        if pretrained_embedding is not None:
            self.embedding = nn.Embedding.from_pretrained(
                torch.tensor(pretrained_embedding, dtype=torch.float32)
            )
        else:
            self.embedding = nn.Embedding(n_vocab, embedding_dim, padding_idx=0)

        self.model = nn.LSTM(embedding_dim, hidden_dim, num_layers=n_layers,
                              bidirectional=bidirectional, dropout=dropout, batch_first=True)
        out_dim = hidden_dim * (2 if bidirectional else 1)
        self.classifier = nn.Linear(out_dim, 1)
        self.dropout = nn.Dropout(dropout)

```

```
def forward(self, input_ids):
    x = self.embedding(input_ids)
    out, _ = self.model(x)
    last = out[:, -1, :]
    last = self.dropout(last)
    return self.classifier(last)    # logits (B,1)
```

## 학습 루프

```
criterion = nn.BCEWithLogitsLoss().to(device)
optimizer = optim.RMSprop(classifier.parameters(), lr=0.001)
# epochs=5, interval=500에서 loss/val acc 로깅
```

## 합성곱 출력 크기(1D/각 차원 동일식)

수식:

$$L_{\text{out}} = \left\lfloor \frac{L_{\text{in}} + 2 \cdot \text{padding} - \text{dilation} \cdot (\text{kernel\_size} - 1) - 1}{\text{stride}} + 1 \right\rfloor$$

예시 계산(본문 예와 동일):

$L_{\text{in}} = 6$ ,  $\text{kernel\_size} = 3$ ,  $\text{stride} = 1$ ,  $\text{padding} = 0$ ,  $\text{dilation} = 1$

$$L_{\text{out}} = \left\lfloor \frac{6 + 0 - 1 \cdot (3 - 1) - 1}{1} + 1 \right\rfloor = \lfloor 4 \rfloor = 4$$

## Conv2d 선언

```
conv = nn.Conv2d(
    in_channels, out_channels,
    kernel_size,      # 예: 3 or (3,3)
    stride=1, padding=0, dilation=1,
    groups=1, bias=True, padding_mode="zeros" # "reflect", "replicate" 가능
)
```

## 용어·하이퍼파라미터

- **from\_pretrained 임베딩**: 외부에서 학습된 임베딩 행렬을 임베딩 층 초기값으로 사용. `<pad>` 는 보통 0벡터(고정), `<unk>` 는 학습 가능하게 두기도 함.
- **BCEWithLogitsLoss**: 시그모이드+BCELoss 결합형. 로짓 입력을 직접 받음.
- **RMSprop( $\eta=0.001$ )**: 지수이동평균으로 스케일을 조절하는 최적화법.
- **Filter/Kernel**: 합성곱 가중치. 크기 예:  $3 \times 3$ ,  $5 \times 5$ .
- **Stride**: 필터 이동 보폭.  $\uparrow \Rightarrow$  출력 공간 크기  $\downarrow$ .
- **Padding**: 입력 테두리에 값(주로 0)을 채움. `padding_mode="zeros|reflect|replicate"`.
- **Channels**: 입력/출력 특성 맵의 개수. `out_channels` = 필터 수.
- **Dilation**: 커널 내 간격 확대(수용영역 $\uparrow$ , 파라미터 수 유지).
- **Activation Map**: 합성곱 출력에 ReLU 등 적용한 맵.

## 핵심 요약

- 합성곱 계층 출력 크기: 커널·패딩·스트라이드·팽창(dilation)에 의해 결정되며 **바림( $\lfloor$ )** 처리.
- 활성화 맵: 합성곱 결과에 주로 **ReLU** 적용해 비선형성 부여.
- 풀링: **최댓값(MaxPool)**· **평균값(AvgPool)**로 특성맵의 공간 크기와 계산량 축소. `AvgPool2d` 는 `count_include_pad` 로 패딩 영역 포함 여부 제어.
- 완전연결(FC): 합성곱/풀링으로 얻은 특성맵을 **Flatten  $\rightarrow$  Linear**로 최종 분류.
- 텍스트에는 2D가 아닌 **1D 합성곱** 사용: 임베딩 차원(E)을 채널로 보고, **여러 커널 크기 (예: 2,3,4)**로 n-gram 패턴을 포착  $\rightarrow$  **Time-wise MaxPool**로 각 필터당 스칼라  $\rightarrow$  **Concat  $\rightarrow$  분류기**.

## 수식 / 코드

### 합성곱/풀링 출력 크기(1D/2D 공통 축별 계산)

$$L_{\text{out}} = \left\lfloor \frac{L_{\text{in}} + 2 \cdot \text{padding} - \text{dilation} \cdot (\text{kernel\_size} - 1) - 1}{\text{stride}} \right\rfloor + 1$$

- 예시(p.343):  $L_{\text{in}} = 6$ ,  $\text{kernel} = 3$ ,  $\text{stride} = 1$ ,  $\text{padding} = 0$ ,  $\text{dilation} = 1$   
 $\Rightarrow L_{\text{out}} = \lfloor (6 - 2 - 1)/1 \rfloor + 1 = 4$

### PyTorch 레이어 시그니처

- `nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, ...)`
- `nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1)`
- `nn.AvgPool2d(kernel_size, stride=None, padding=0, count_include_pad=True)`

## 텍스트용 1D-CNN 형상 관찰

- 임베딩 후 텐서: (B, T, E) → `permute(0,2,1)` 로 (B, E, T)
- `nn.Conv1d(in_channels=E, out_channels=F, kernel_size=k)` → (B, F, T-k+1)
- `nn.MaxPool1d(kernel_size=T-k+1)` → (B,F,1)(B, F, 1)(B,F,1) → `squeeze(-1)` 로 (B, F)
- 여러 k의 출력을 `torch.cat` → Linear(들) → 로짓

간단 스니펫:

```
emb = self.embedding(inputs)          # (B,T,E)
x = emb.permute(0, 2, 1)              # (B,E,T)
feat_list = [pool(conv(x)).squeeze(-1) for conv, pool in blocks] # (B,F)
h = torch.cat(feat_list, dim=1)        # (B, sum_F)
logits = self.classifier(self.dropout(h)) # (B, 1)
```

## 텍스트 CNN(1D Conv)

- 텍스트는 2D가 아닌 1D 합성곱을 적용: 임베딩 벡터의 차원은 **채널(C)**, 토큰 순서 길이는 **시퀀스 길이(L)**로 보고, **가로(수평) 방향**으로만 필터를 움직입니다.
- 필터 크기 **k(3·4·5 ...)** 는 n-gram 탐지기로 동작. Conv1d → **ReLU** → **전역(Max) 풀링**으로 각 필터에서 **스칼라 1개**를 얻고, 여러 필터의 스칼라를 **concat** → **FC**로 분류합니다.
- **사전학습 임베딩**(Word2Vec 등)으로 `nn.Embedding.from_pretrained(...)` 초기화 후 사용.
- 구현 포인트: 입력 (N, L) → 임베딩 (N, L, D) → **permute** → Conv1d 입력형 (N, D, L) 맞추기.
- 학습: **BCEWithLogitsLoss**(이진 분류), 최적화 **Adam**, 결과 예: **Val Acc ≈ 0.80** 수준.

## 수식 / 코드 핵심

## 1) 출력 길이(1D Conv)

$$L_{\text{out}} = \left\lfloor \frac{L_{\text{in}} + 2 \cdot \text{padding} - \text{dilation} \cdot (k - 1) - 1}{\text{stride}} + 1 \right\rfloor$$

- 텍스트 CNN 기본 설정: `stride=1`, `padding=0`, `dilation=1`  $\rightarrow L_{\text{out}} = L_{\text{in}} - k + 1$

## 2) 전역(Max) 풀링 크기

- 각 필터별 시퀀스 축 전체를 한 번에 줄이는 전역 풀링:

$$\text{kernel\_size}^{\text{pool}} = L_{\text{out}} = L_{\text{in}} - k + 1$$

$\rightarrow$  출력 shape: `(N, out_channels=1, 1)`  $\rightarrow$  `squeeze(-1)` 로 `(N, 1)`.

## 3) 모듈 구성

```
# 임베딩: 사전학습 행렬로 초기화
self.embedding = nn.Embedding.from_pretrained(
    torch.tensor(pretrained_embedding, dtype=torch.float32)
    # freeze=True(기본)  $\rightarrow$  미세조정하려면 freeze=False
)
D = self.embedding.weight.shape[1] # embedding_dim

# 다중 필터(예: [3,3,4,4,5,5])
convs = []
for k in filter_sizes:
    convs.append(nn.Sequential(
        nn.Conv1d(in_channels=D, out_channels=1, kernel_size=k, stride=1, padding=0),
        nn.ReLU(),
        nn.MaxPool1d(kernel_size=max_length - k + 1) # 전역 풀링
    ))
self.conv_filters = nn.ModuleList(convs)

# 분류 헤드
self.pre_classifier = nn.Linear(len(filter_sizes), len(filter_sizes))
self.dropout = nn.Dropout(p=0.5)
self.classifier = nn.Linear(len(filter_sizes), 1)

def forward(inputs):          # inputs: (N, L)
```

```

x = self.embedding(inputs)    # (N, L, D)
x = x.permute(0, 2, 1)        # (N, D, L) ← Conv1d 기대형
feats = [m(x).squeeze(-1) for m in self.conv_filters] # 각 (N,1) 리스트
h = torch.cat(feats, dim=1)    # (N, #filters)
h = self.pre_classifier(h)
h = self.dropout(h)
logits = self.classifier(h)   # (N, 1)
return logits

```

학습 루틴:

```
criterion = nn.BCEWithLogitsLoss()
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

## 용어 · 하이퍼파라미터

- **Conv1d**: (N, C<sub>in</sub>, L) 입력에 커널 폭 **k**(토큰 개수)로 시퀀스 축을 따라 합성곱.
- **Filter size(k)**: 3/4/5 등 n-gram 길이. 여러 k를 병렬로 두어 다양한 패턴 포착.
- **Channel(C)**: 임베딩 차원 D가 **입력 채널 수**로 매핑.
- **Global Max Pooling**: 각 필터의 가장 강한 활성 하나만 남겨 (**스파스·해석 쉬움**).
- **Dropout(0.5)**: 과적합 방지.
- **from\_pretrained**: 사전학습 임베딩으로 초기화. 기본 `freeze=True` (미세조정 원하면 `freeze=False` ).
- **손실/옵티마이저**: `BCEWithLogitsLoss` (시그모이드+CE 결합), `Adam(lr=0.001)` .