# 5장
# 단어와 타입 임베딩

발표자 : 최환규

# 목차

- 임베딩을 배우는 이유
- 실습
  - CBOW 임베딩 학습하기
  - 문서 분류에 사전 훈련된 임베딩을 사용한 전이 학습 (fine-tuning)

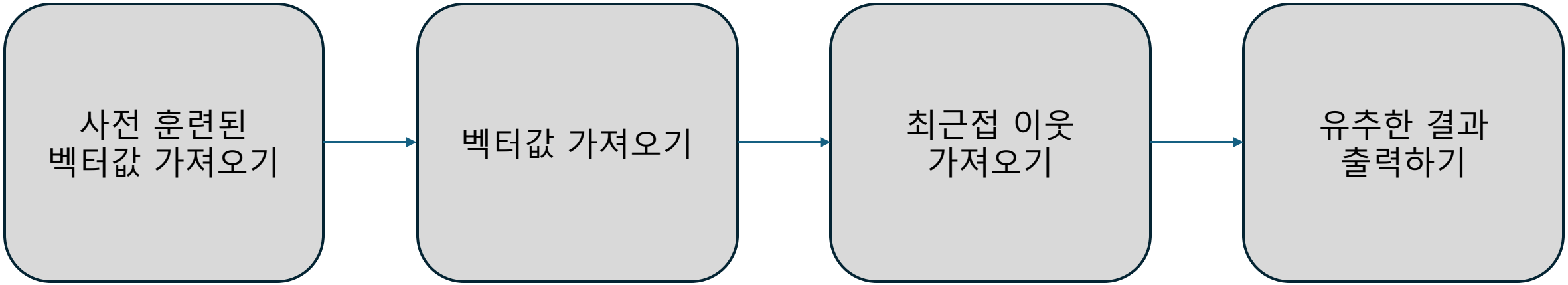
# 임베딩(embedding) 이란?

- 자연어 처리와 기계 학습 분야에서 텍스트나 범주형 데이터를 수치형 벡터로 변환하는 기술
- 주로 사용되는 기술로 word embedding이 있음.
  - 단어를 고정된 차원의 실수 벡터로 매핑하는 방법으로, 단어 간의 의미적 유사성을 보존하려는 목적
- 대표적인 워드 임베딩 모델
  - Word2Vec
  - GloVe
  - FastText
  - BERT

# 임베딩을 배우는 이유

- 의미적 유사성 표현 : 비슷한 의미를 가진 단어들이 서로 가까이 위치함
- 문맥 파악 : 임베딩은 문맥을 고려해 단어를 벡터로 표현 가능하기 때문
- 차원 감소 : 데이터를 효율적이게 다룰 수 있음.

# 실습 : Glove 사용하기.



# 사전훈련된 임베딩값 불러오기

- 사전 훈련된 임베딩 값을 객체로 생성해주는 부분.
  (인덱스, 벡터) 형식으로 반환.

```python
@classmethod
def from_embeddings_file(cls, embedding_file):
    """사전 훈련된 벡터 파일에서 객체를 만듭니다.

    벡터 파일은 다음과 같은 포맷입니다:
        word0 x0_0 x0_1 x0_2 x0_3 ... x0_N
        word1 x1_0 x1_1 x1_2 x1_3 ... x1_N

    매개변수:
        embedding_file (str): 파일 위치
    반환값:
        PretrainedEmbeddings의 인스턴스
    """
    word_to_index = {}
    word_vectors = []

    with open(embedding_file, encoding='utf-8') as fp:
        for line in fp.readlines():
            line = line.split(" ")
            word = line[0]
            vec = np.array([float(x) for x in line[1:]])

            word_to_index[word] = len(word_to_index)
            word_vectors.append(vec)

    return cls(word_to_index, word_vectors)
```

코드1-1 사전훈련된 임베딩 객체 생성

# 벡터값 가져오기

```
def get_embedding(self, word):
    """
    매개변수:
        word (str)
    반환값
        임베딩 (numpy.ndarray)
    """
    return self.word_vectors[self.word_to_index[word]]
```

코드1-2 벡터 값 가져오기

- 사전 학습된 임베딩 값을 반환하는 함수 코드.

# 최근접 이웃 가져오기

```python
def get_closest_to_vector(self, vector, n=1):
    """벡터가 주어지면 n 개의 최근접 이웃을 반환합니다
    매개변수:
        vector (np.ndarray): Annoy 인덱스에 있는 벡터의 크기와 같아야 합니다
        n (int): 반환될 이웃의 개수
    반환값:
        [str, str, ...]: 주어진 벡터와 가장 가까운 단어
            단어는 거리순으로 정렬되어 있지 않습니다.
    """
    nn_indices = self.index.get_nns_by_vector(vector, n)
    return [self.index_to_word[neighbor] for neighbor in nn_indices]
```

코드1-3 최근접 이웃 가져오기

- 최근접 이웃을 반환하는 함수.
- Annoyindex의 메소드를 사용

# 유추한 결과 출력하기

- 추론방식
  - Word1의 벡터와 word2의 벡터 사이의 거리를 이용해 word4의 벡터를 구합니다.
  - 코드 1-3의 함수에 word4의 벡터를 대입하고 반환 받은 값을 형식에 맞게 출력합니다.
- 입력 방식
  - (blue ,color ,dog)
- 출력 방식
  - blue : color :: dog : animal

```python
def compute_and_print_analogy(self, word1, word2, word3):
    """단어 임베딩을 사용한 유추 결과를 출력합니다

    word1이 word2일 때 word3은 __입 실습 : Glove 사용하기.
    이 메서드는 word1 : word2 :: word3 : word4를 출력합니다

    매개변수:
        word1 (str)
        word2 (str)
        word3 (str)
    """
    vec1 = self.get_embedding(word1)
    vec2 = self.get_embedding(word2)
    vec3 = self.get_embedding(word3)

    # 네 번째 단어 임베딩을 계산합니다
    spatial_relationship = vec2 - vec1
    vec4 = vec3 + spatial_relationship

    closest_words = self.get_closest_to_vector(vec4, n=4)
    existing_words = set([word1, word2, word3])
    closest_words = [word for word in closest_words
                     if word not in existing_words]

    if len(closest_words) == 0:
        print("계산된 벡터와 가장 가까운 이웃을 찾을 수 없습니다!")
        return

    for word4 in closest_words:
        print("{} : {} :: {} : {}".format(word1, word2, word3, word4))
```

코드1-4 유추한 결과 출력.

# 임베딩에서 발생가능한 문제.

- 단어 벡터의 성질로 인한 잘못된 관계 형성.
  - 발생할 수 있는 문제, 항상 그렇지는 않음

```
1 embeddings.compute_and_print_analogy('cat', 'kitten', 'dog')
```

```
cat : kitten :: dog : adorable
cat : kitten :: dog : sassy
cat : kitten :: dog : furry
```

```
1 embeddings.compute_and_print_analogy('blue', 'color', 'dog')
```

```
blue : color :: dog : breed
blue : color :: dog : pig
blue : color :: dog : bites
blue : color :: dog : mouse
```

```
1 embeddings.compute_and_print_analogy('leg', 'legs', 'hand')
```

```
leg : legs :: hand : hands
leg : legs :: hand : eyes
```

```
1 embeddings.compute_and_print_analogy('toe', 'foot', 'finger')
```

```
toe : foot :: finger : attached
toe : foot :: finger : broken
toe : foot :: finger : inside
```

코드 1-5 단어 임베딩을 이용한 유추작업

# 임베딩에서 발생가능한 문제.

- 언어 규칙과 성문화된 문화 편견 구별이 어려움.
  - 임베딩의 편향성을 염두에 두고 작업해야함.

```
1 embeddings.compute_and_print_analogy('blue', 'democrat', 'red')
```

```
blue : democrat :: red : republican
blue : democrat :: red : congressman
blue : democrat :: red : senator
```

```
1 embeddings.compute_and_print_analogy('man', 'king', 'woman')
```

```
man : king :: woman : queen
man : king :: woman : throne
man : king :: woman : prince
```

```
1 embeddings.compute_and_print_analogy('man', 'doctor', 'woman')
```

```
man : doctor :: woman : nurse
man : doctor :: woman : physician
man : doctor :: woman : pregnant
```

코드 1-6 단어 임베딩을 이용한 유추작업

# 실습: CBOW임베딩 학습하기

- 학습에 사용된 데이터셋 소개
- CBOW 학습을 위한 데이터셋 생성
- Vocabulary, Vectorizer 클래스 소개
- 분류모델 소개
  - 모델 훈련
  - 모델 한계점

# 학습에 사용한 dataset

- 메리 셸리의 소설 프랑켄슈타인
- 약 7만개의 단어로 구성됨.
- 기본적인 전처리 구두점제거, 소문자로 변환 공백 기준으로 토큰화

**The Project Gutenberg eBook of Frankenstein, by Mary Wollstonecraft Shelley**

This eBook is for the use of anyone anywhere in the United States and most other parts of the world at no cost and with almost no restrictions whatsoever. You may copy it, give it away or re-use it under the terms of the Project Gutenberg License included with this eBook or online at www.gutenberg.org. If you are not located in the United States, you will have to check the laws of the country where you are located before using this eBook.

Title: Frankenstein
or, The Modern Prometheus

Author: Mary Wollstonecraft Shelley

Release Date: October 31, 1993 [eBook #84]
[Most recently updated: December 2, 2022]

Language: English

Character set encoding: UTF-8

Produced by: Judith Boss, Christy Phillips, Lynn Hanninen and David Meltzer. HTML version by Al Haines.
Further corrections by Menno de Leeuw.

*** START OF THE PROJECT GUTENBERG EBOOK FRANKENSTEIN ***

프랑켄슈타인 ebook webpage

# CBOW 학습을 위한 데이터셋 생성

```
# Create windows
flatten = lambda outer_list: [item for inner_list in outer_list for item in inner_list]
windows = flatten([list(nltk.ngrams([MASK_TOKEN] * args.window_size + ₩
                                    sentence.split(' ') + [MASK_TOKEN] * ₩
                                    args.window_size, args.window_size * 2 + 1)) ₩
    for sentence in tqdm_notebook(cleaned_sentences)])

# Create cbow data
data = []
for window in tqdm_notebook(windows):
    target_token = window[args.window_size]
    context = []
    for i, token in enumerate(window):
        if token == MASK_TOKEN or i == args.window_size:
            continue
        else:
            context.append(token)
    data.append([' '.join(token for token in context), target_token])


# Convert to dataframe
cbow_data = pd.DataFrame(data, columns=["context", "target"])
```

코드2-1 윈도우 생성하기

# CBOW 학습을 위한 데이터셋 표현

| 전처리된 문장 | I pitied Frankenstein my pity amounted to horror I abhorred myself |
|---|---|
| 윈도 #1 | I pitied Frankenstein my pity amounted to horror I abhorred myself |
| 윈도 #2 | I pitied Frankenstein my pity amounted to horror I abhorred myself |
| 윈도 #3 | I pitied Frankenstein my pity amounted to horror I abhorred myself |
| 윈도 #4 | I pitied Frankenstein my pity amounted to horror I abhorred myself |

# Vocabulary class

- 변경된 데이터 셋에 따라 변화
  - MASK 토큰 추가
    (학습에서 제외시킬 부분)

```python
def __init__(self, token_to_idx=None, mask_token="<MASK>", add_unk=True,
             unk_token="<UNK>"):
    """
    매개변수:
        token_to_idx (dict): 기존 토큰-인덱스 매핑 딕셔너리
        mask_token (str): Vocabulary에 추가할 MASK 토큰.
            모델 파라미터를 업데이트하는데 사용하지 않는 위치를 나타냅니다.
        add_unk (bool): UNK 토큰을 추가할지 지정하는 플래그
        unk_token (str): Vocabulary에 추가할 UNK 토큰
    """

    if token_to_idx is None:
        token_to_idx = {}
    self._token_to_idx = token_to_idx

    self._idx_to_token = {idx: token
                          for token, idx in self._token_to_idx.items()}

    self._add_unk = add_unk
    self._unk_token = unk_token
    self._mask_token = mask_token

    self.mask_index = self.add_token(self._mask_token)
    self.unk_index = -1
    if add_unk:
        self.unk_index = self.add_token(unk_token)
```

코드2-2 vocabulary class

# Vectorizer class

- 문맥의 토큰수가 최대 길이보다 적으면 나머지 항목은 mask_index로 채워짐.

```
def vectorize(self, context, vector_length=-1):
    """
    매개변수:
        context (str): 공백으로 나누어진 단어 문자열
        vector_length (int): 인덱스 벡터의 길이 매개변수
    """

    indices = [self.cbow_vocab.lookup_token(token) ₩
               for token in context.split(' ')]
    if vector_length < 0:
        vector_length = len(indices)

    out_vector = np.zeros(vector_length, dtype=np.int64)
    out_vector[:len(indices)] = indices
    out_vector[len(indices):] = self.cbow_vocab.mask_index

    return out_vector
```

코드2-3 Vectorizer class

# 분류 모델

- Ebedding 층을 사용해 문맥의 단어를 나타내는 인덱스를 각 단어에 대한 벡터로 만듭니다.
- 전반적인 문맥을 감지하도록 벡터를 결합
  - 이 모델에서는 벡터 결합 방법이 벡터를 더해주는 것입니다.
- Linear층에서 문맥 벡터를 사용해 예측 벡터 계산
  - 예측 벡터 : 전체 어휘 사전에 대한 확률 분포.
    가장 큰 값이 타깃 단어에 대한 예측을 나타냄

```
class CBOWClassifier(nn.Module): # Simplified cbow Model
    def __init__(self, vocabulary_size, embedding_size, padding_idx=0):
        """
        매개변수:
            vocabulary_size (int): 어휘 사전 크기,
                임베딩 개수와 예측 벡터 크기를 결정합니다
            embedding_size (int): 임베딩 크기
            padding_idx (int): 기본값 0; 임베딩은 이 인덱스를 사용하지 않습니다
        """
        super(CBOWClassifier, self).__init__()

        self.embedding =  nn.Embedding(num_embeddings=vocabulary_size,
                                       embedding_dim=embedding_size,
                                       padding_idx=padding_idx)
        self.fc1 = nn.Linear(in_features=embedding_size,
                             out_features=vocabulary_size)
```

코드 2-4 classifier class

```
def forward(self, x_in, apply_softmax=False):
    """분류기의 정방향 계산

    매개변수:
        x_in (torch.Tensor): 입력 데이터 텐서
            x_in.shape는 (batch, input_dim)입니다.
        apply_softmax (bool): 소프트맥스 활성화 함수를 위한 플래그
            크로스-엔트로피 손실을 사용하려면 False로 지정합니다
    반환값:
        결과 텐서. tensor.shape은 (batch, output_dim)입니다.
    """
    x_embedded_sum = F.dropout(self.embedding(x_in).sum(dim=1), 0.3)
    y_out = self.fc1(x_embedded_sum)

    if apply_softmax:
        y_out = F.softmax(y_out, dim=1)

    return y_out
```

코드 2-5 classifier class 정방향 계산

# 모델 훈련

- 레스토랑 리뷰 감성 분류기 예제와 같은 부분으로 생략.

```
args = Namespace(
    # 날짜와 경로 정보
    cbow_csv="data/books/frankenstein_with_splits.csv",
    vectorizer_file="vectorizer.json",
    model_state_file="model.pth",
    save_dir="model_storage/ch5/cbow",
    # 모델 하이퍼파라미터
    embedding_size=50,
    # 훈련 하이퍼파라미터
    seed=1337,
    num_epochs=100,
    learning_rate=0.0001,
    batch_size=32,
    early_stopping_criteria=5,
    # 실행 옵션
    cuda=True,
    catch_keyboard_interrupt=True,
    reload_from_files=False,
    expand_filepaths_to_save_dir=True
)
```

코드 2-6 CBOW 훈련 매개변수

# 실습의 한계점

```
print("테스트 손실: {};".format(train_state['test_loss']))
print("테스트 정확도: {}".format(train_state['test_acc']))
```

```
테스트 손실: 7.676548838895911;
테스트 정확도: 13.191176470588225
```

코드 2-7 CBOW 테스트 결과

- 간략화 된 모델
  -> 성능이 좋지 않다.
- 적은 데이터양
  -> 많은 규칙성을 감지하기엔 데이터가 부족하다.

# 실습: 문서 분류에 사전 훈련된 임베딩을 사용한 전이 학습

- 텍스트 분류에서 사전 훈련된 단어 임베딩의 효과 확인
- 기사 제목에 초점을 맞춰 카테고리를 예측하는 다중 분류 작업

# 학습에 사용한 dataset

- AG_news corpus
  - 2005년에 수집한 뉴스 기사 모음

You are not authorized to change the corpus or to re-distribute (part of) it with a different name.

- db version of the AG news article's corpus
- xml version of the AG news article's corpus (thanks to Paolo Ferragina)

**DB Table**

| Field | Type | Null | Key | Default | Extra |
|---|---|---|---|---|---|
| source | varchar(32) | | PRI | | |
| url | varchar(255) | | PRI | | |
| title | text | YES | MUL | NULL | |
| image | varchar(255) | YES | | NULL | |
| category | varchar(32) | | PRI | | |
| description | text | YES | | NULL | |
| rank | int(11) | YES | | NULL | |
| pubdate | timestamp | YES | | CURRENT_TIMESTAMP | |
| video | varchar(255) | YES | | NULL | |

# Dataset class

```
def __init__(self, news_df, vectorizer):
    """
    매개변수:
        news_df (pandas.DataFrame): 데이터셋
        vectorizer (NewsVectorizer): 데이터셋에서 만든 NewsVectorizer 객체
    """
    self.news_df = news_df
    self._vectorizer = vectorizer

    # +1 if only using begin_seq, +2 if using both begin and end seq tokens
    measure_len = lambda context: len(context.split(" "))
    self._max_seq_length = max(map(measure_len, news_df.title)) + 2
```

코드 3-1 Dataset class

- Begin_seq, end_seq토큰을 둘다 사용하기 때문에 +2

# Dataset class

```
# 클래스 가중치
class_counts = news_df.category.value_counts().to_dict()
def sort_key(item):
    return self._vectorizer.category_vocab.lookup_token(item[0])
sorted_counts = sorted(class_counts.items(), key=sort_key)
frequencies = [count for _, count in sorted_counts]
self.class_weights = 1.0 / torch.tensor(frequencies, dtype=torch.float32)
```

코드 3-2 Dataset class 가중치

- 데이터의 불균형을 고려하여 클래스에 가중치를 부여

# Vocabulary class

- Vocabulary 클래스를 상속해 Sequence Vocabulary를 제작
  - UNK, MASK, BEGIN-OF-SEQUENCE, END-OF-SEQUENCE 4개의 토큰을 사용.

```
def __init__(self, token_to_idx=None, unk_token="<UNK>",
             mask_token="<MASK>", begin_seq_token="<BEGIN>",
             end_seq_token="<END>"):

    super(SequenceVocabulary, self).__init__(token_to_idx)

    self._mask_token = mask_token
    self._unk_token = unk_token
    self._begin_seq_token = begin_seq_token
    self._end_seq_token = end_seq_token

    self.mask_index = self.add_token(self._mask_token)
    self.unk_index = self.add_token(self._unk_token)
    self.begin_seq_index = self.add_token(self._begin_seq_token)
    self.end_seq_index = self.add_token(self._end_seq_token)

def to_serializable(self):
    contents = super(SequenceVocabulary, self).to_serializable()
    contents.update({'unk_token': self._unk_token,
                     'mask_token': self._mask_token,
                     'begin_seq_token': self._begin_seq_token,
                     'end_seq_token': self._end_seq_token})
    return contents
```

코드 3-3 Vocabulary class

# Sequence Vocabulary 벡터 변환 예제

Sequence Vocabulary
{
'<MASK>' : 0,
'<UNK>' : 1,
'<BEGIN-OF-SEQUENCE>': 2,
'<END-OF-SEQUENCE>' : 3,
'is' : 4,
'happy' : 5
}
Max_seq=6

"Jerry is happy"

| 1 | 4 | 5 | | | |
|---|---|---|---|---|---|

| 2 | 1 | 4 | 5 | 3 | |
|---|---|---|---|---|---|

| 2 | 1 | 4 | 5 | 3 | 0 |
|---|---|---|---|---|---|

0단계 : 문장을 입력 받습니다.

1단계 : 단어를 정수로 매핑합니다.

2단계 : 문장을 경계 토큰으로 감쌉니다.

3단계 : 모든 벡터의 길이가 같도록 0으로 패딩합니다.

# Vectorizer class

- 시작 부분을 begin_seq_index, 끝 부분을 end_seq_index로 감싸줍니다.
- 문맥의 토큰수가 최대 길이보다 적으면 나머지 항목은 mask_index로 채워집니다.

```
def vectorize(self, title, vector_length=-1):
    """
    매개변수:
        title (str): 공백으로 나누어진 단어 문자열
        vector_length (int): 인덱스 벡터의 길이 매개변수
    반환값:
        벡터로 변환된 제목 (numpy.array)
    """
    indices = [self.title_vocab.begin_seq_index]
    indices.extend(self.title_vocab.lookup_token(token)
                   for token in title.split(" "))
    indices.append(self.title_vocab.end_seq_index)

    if vector_length < 0:
        vector_length = len(indices)

    out_vector = np.zeros(vector_length, dtype=np.int64)
    out_vector[:len(indices)] = indices
    out_vector[len(indices):] = self.title_vocab.mask_index

    return out_vector
```

코드 3-4 Vectorizer class

# 사전 훈련된 임베딩 값 불러오기.

```python
def load_glove_from_file(glove_filepath):
    """GloVe 임베딩 로드

    매개변수:
        glove_filepath (str): 임베딩 파일 경로
    반환값:
        word_to_index (dict), embeddings (numpy.ndarary)
    """

    word_to_index = {}
    embeddings = []
    with open(glove_filepath, "r") as fp:
        for index, line in enumerate(fp):
            line = line.split(" ") # each line: word num1 num2 ...
            word_to_index[line[0]] = index # word = line[0]
            embedding_i = np.array([float(val) for val in line[1:]])
            embeddings.append(embedding_i)
    return word_to_index, np.stack(embeddings)
```

코드 3-5 Glove 임베딩 로드

- 사전 훈련된 임베딩을 로드한 후, 실제 데이터에 있는 단어에 해당하는 임베딩의 일부를 선택하여 반환

# 사전 훈련된 임베딩 값 지정하기

```python
def make_embedding_matrix(glove_filepath, words):
    """
    특정 단어 집합에 대한 임베딩 행렬을 만듭니다.

    매개변수:
        glove_filepath (str): 임베딩 파일 경로
        words (list): 단어 리스트
    """
    word_to_idx, glove_embeddings = load_glove_from_file(glove_filepath)
    embedding_size = glove_embeddings.shape[1]

    final_embeddings = np.zeros((len(words), embedding_size))

    for i, word in enumerate(words):
        if word in word_to_idx:
            final_embeddings[i, :] = glove_embeddings[word_to_idx[word]]
        else:
            embedding_i = torch.ones(1, embedding_size)
            torch.nn.init.xavier_uniform_(embedding_i)
            final_embeddings[i, :] = embedding_i

    return final_embeddings
```

코드 3-6 임베딩 지정

- Embedding 층의 가중치 행렬을 코드 3-5에서 선택한 임베딩으로 임베딩 행렬을 제작하여 반환

# 분류 모델

```
super(NewsClassifier, self).__init__()

if pretrained_embeddings is None:

    self.emb = nn.Embedding(embedding_dim=embedding_size,
                            num_embeddings=num_embeddings,
                            padding_idx=padding_idx)
else:
    pretrained_embeddings = torch.from_numpy(pretrained_embeddings).float()
    self.emb = nn.Embedding(embedding_dim=embedding_size,
                            num_embeddings=num_embeddings,
                            padding_idx=padding_idx,
                            _weight=pretrained_embeddings)
```

코드 3-7 임베딩 교체

- 성씨 분류기에서 사용한 합성곱 분류기를 기반으로 제작
- 임베딩 층의 가중치를 사전 훈련된 임베딩으로 교체.
- 임베딩 층 외에는 CNN으로 성씨 분류하기 예제와 동일함.

# 모델 훈련

- 중복되는 부분으로 생략

```
args = Namespace(
    # 날짜와 경로 정보
    news_csv="data/ag_news/news_with_splits.csv",
    vectorizer_file="vectorizer.json",
    model_state_file="model.pth",
    save_dir="model_storage/ch5/document_classification",
    # 모델 하이퍼파라미터
    glove_filepath='D:₩대학공부₩동계인턴₩nlp-with-pytorch-main₩chapter_5₩data₩glove.6B',
    use_glove=False,
    embedding_size=100,
    hidden_dim=100,
    num_channels=100,
    # 훈련 하이퍼파라미터
    seed=1337,
    learning_rate=0.001,
    dropout_p=0.1,
    batch_size=128,
    num_epochs=100,
    early_stopping_criteria=5,
    # 실행 옵션
    cuda=True,
    catch_keyboard_interrupt=True,
    reload_from_files=False,
    expand_filepaths_to_save_dir=True
)
```

코드 3-8 훈련 매개변수

# 모델 평가

```
1 print("테스트 손실: {};".format(train_state['test_loss']))
2 print("테스트 정확도: {}".format(train_state['test_acc']))
```

```
테스트 손실: 0.6973735798682495;
테스트 정확도: 78.78348214285714
```

코드 3-9 테스트 데이터 평가

- 사전 훈련된 임베딩을 사용하니 성능이 나쁘지 않음.

# 새로운 뉴스 제목의 카테고리 예측하기

- 새로운 뉴스 제목의 카테고리를 예측합니다.

```
def predict_category(title, classifier, vectorizer, max_length):
    """뉴스 제목을 기반으로 카테고리를 예측합니다

    매개변수:
        title (str): 원시 제목 문자열
        classifier (NewsClassifier): 훈련된 분류기 객체
        vectorizer (NewsVectorizer): 해당 Vectorizer
        max_length (int): 최대 시퀀스 길이
            노트: CNN은 입력 텐서 크기에 민감합니다.
                훈련 데이터처럼 동일한 크기를 갖도록 만듭니다.
    """
    title = preprocess_text(title)
    vectorized_title = \
        torch.tensor(vectorizer.vectorize(title, vector_length=max_length))
    result = classifier(vectorized_title.unsqueeze(0), apply_softmax=True)
    probability_values, indices = result.max(dim=1)
    predicted_category = vectorizer.category_vocab.lookup_index(indices.item())

    return {'category': predicted_category,
            'probability': probability_values.item()}
```

코드 3-10 카테고리 예측하기