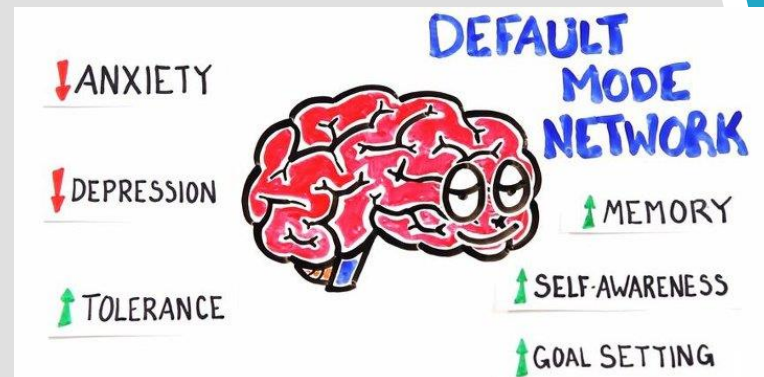
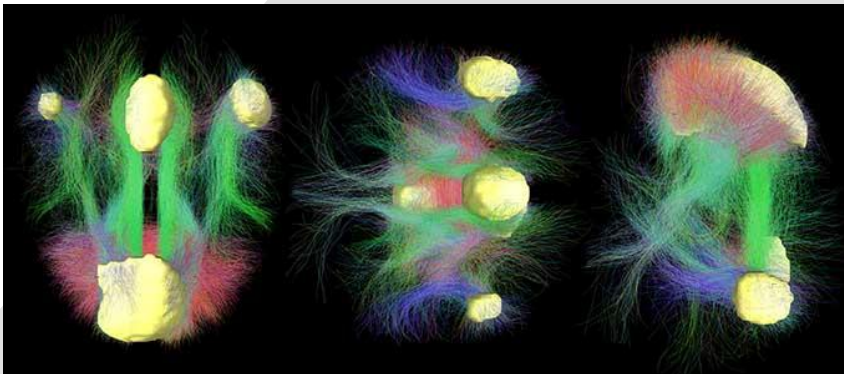


6장 *Deep learning for text and sequences*

“default mode network”





This chapter covers



- Preprocessing text data into useful representations – **tokens to vectors**
- Working with **recurrent neural networks**
- Using 1D convnets for **sequence processing**

○○○ *This chapter covers* ○○○

- ▶ sequences of **word**, **timeseries**, and sequence **data** in general
 - ▶ *recurrent neural networks* and *1D convnets*
 - ▶ Applications of these algorithms include the following:
 - Document classification - identifying the **topic of an article** or the **author of a book**
 - Sequence-to-sequence learning - **English sentence into French**
 - Sentiment analysis - classifying the sentiment of **tweets** or **movie reviews** as positive or negative
 - Timeseries forecasting - predicting the **future weather** given recent weather data
1. **sentiment** analysis on the IMDB dataset
 2. **temperature** forecasting.

6.1 Working with text data

- ▶ **natural-language** understanding - document classification, sentiment analysis, author identification, and even question-answering (QA)
- ▶ Deep learning for natural-language processing is **pattern recognition** applied to words, sentences, and paragraphs
- ▶ **Vectorizing text** is the process of transforming **text** into **numeric tensors**.
 - ▶ Segment text into **words**, and transform each word into a **vector**.
 - ▶ Segment text into **characters**, and transform each character into a **vector**.
 - ▶ Extract **n-grams** of words or characters, and transform each n-gram into a **vector**.
- ▶ **N-grams** are overlapping groups of **multiple consecutive words or characters**.
- ▶ **tokens** - break down text (**words, characters, or n-grams**)
- ▶ **tokenization** - breaking text into such tokens
- ▶ two major ones: **one-hot encoding** of tokens, and **token embedding**

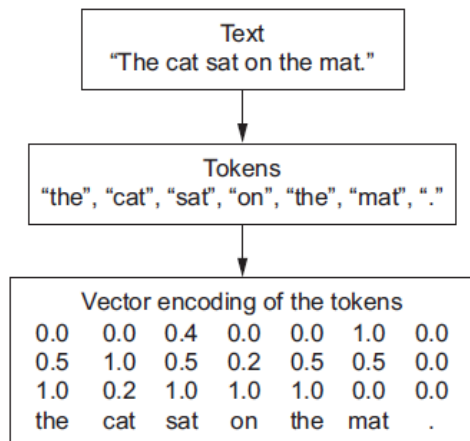


Figure 6.1 From text to tokens to vectors

6.1 Working with text data

Understanding **n-grams** and **bag-of-words (BoW)**

▶ **Word n-grams** are groups of N (or fewer) consecutive words that you can extract from a sentence.

▶ “The cat sat on the mat.” – **set of 2-grams**:

`{"The", "The cat", "cat", "cat sat", "sat",
"sat on", "on", "on the", "the", "the mat", "mat"}` **bag-of-2-grams**

▶ It may also be decomposed into the following set of **3-grams**:

`{"The", "The cat", "cat", "cat sat", "The cat sat",
"sat", "sat on", "on", "cat sat on", "on the", "the", "sat
on the", "the mat", "mat", "on the mat"}` **bag-of-3-grams**

- ▶ Because bag-of-words **isn't an order-preserving tokenization method** (the tokens generated are understood as a **set**, not a sequence, and the general structure of the sentences is lost)
- ▶ unavoidable **feature-engineering** tool when using lightweight, shallow text-processing models such as **logistic regression** and **random forests**.

6.1 Working with text data

6.1.1 One-hot encoding of words and characters

- ▶ **One-hot encoding** - turn a token into a vector
- ▶ **IMDB** and **Reuters** examples - done with words
- ▶ a unique **integer index** with every word - **binary vector** of size N (the size of the vocabulary)
- ▶ one-hot encoding can be done at the **character** level

6.1 Working with text data

Listing 6.1 Word-level one-hot encoding (toy example)

```
import numpy as np
samples = ['The cat sat on the mat.', 'The dog ate my homework.']
# 데이터에 있는 모든 토큰의 인덱스를 구축합니다
token_index = {} # dictionary
for sample in samples:
    # split() 메서드를 사용해 샘플을 토큰으로 나눕니다.
    for word in sample.split():
        if word not in token_index:
            token_index[word] = len(token_index) + 1
            # 인덱스 0은 사용하지 않습니다.
# {'The': 1, 'cat': 2, 'sat': 3, 'on': 4, 'the': 5, 'mat.': 6, 'dog': 7, 'ate': 8, 'my': 9, 'homework.': 10}
# 샘플을 벡터로 변환
max_length = 10
results = np.zeros((len(samples), max_length, max(token_index.values()) + 1))
for i, sample in enumerate(samples):
    for j, word in list(enumerate(sample.split()))[:max_length]:
        index = token_index.get(word)
        results[i, j, index] = 1.
```

[[0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]	The	[[0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]	The
[0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]	cat	[0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]	dog
[0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]	sat	[0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]	ate
[0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]	on	[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]	my
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]	the	[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]	homework
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]	mat	[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]	
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]		[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]	
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]		[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]	
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]		[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]	
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]		[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]	

6.1 Working with text data

Listing 6.2 Character-level one-hot encoding (toy example)

```
import string
samples = ['The cat sat on the mat.', 'The dog ate my homework.']
characters = string.printable # 출력 가능한 모든 ASCII 문자
token_index = dict(zip(characters, range(1, len(characters) + 1)))
max_length = 50
```

```
results = np.zeros((len(samples), max_length,
                    max(token_index.values())+1))
for i, sample in enumerate(samples):
    for j, character in enumerate(sample[:max_length]):
        index = token_index.get(character)
        results[i, j, index] = 1.
```

{'0': 1, '1': 2, '2': 3, '3': 4, '4': 5, '5': 6, '6': 7, '7': 8, '8': 9, '9': 10, 'a': 11, 'b': 12, 'c': 13, 'd': 14, 'e': 15, 'f': 16, 'g': 17, 'h': 18, 'i': 19, 'j': 20, 'k': 21, ..., 'A': 37, 'B': 38, ..., '\x0c': 100}

```
[[[0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] ... [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.]] 'The cat sat on the mat.'
[[0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] ... [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.]] 'The dog ate my homework.'
```


6.1 Working with text data

Listing 6.3 Using Keras for word-level one-hot encoding

```
from keras.preprocessing.text import Tokenizer
samples = ['The cat sat on the mat.', 'The dog ate my homework.']
# 가장 빈도가 높은 1,000개의 단어만 선택하도록 Tokenizer 객체를 만듭니다.
tokenizer = Tokenizer(num_words=1000)
# Turns strings into lists of integer indices by word_index
tokenizer.fit_on_texts(samples) # 입력에 맞게 내부의 word_index를 만드는 함수
# tokenizer.word_index = {'the': 1, 'cat': 2, 'sat': 3, 'on': 4, 'mat': 5, 'dog': 6, 'ate': 7, 'my': 8, 'homework': 9}

# Turns strings into lists of integer indices
sequences = tokenizer.texts_to_sequences(samples)
# Sequences = [[1, 2, 3, 4, 1, 5], [1, 6, 7, 8, 9]]

# directly get the one-hot binary representations.
# Vectorization modes other than one-hot encoding are supported by this tokenizer!
one_hot_results = tokenizer.texts_to_matrix(samples, mode='binary')
# one_hot_results = [[0. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. ... 0. 0. 0.]
#                    [0. 1. 0. 0. 0. 0. 1. 1. 1. 1. 0. ... 0. 0. 0.]]

# 계산된 단어 인덱스를 구합니다.
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
# Found 9 unique tokens.
```

6.1 Working with text data

- ▶ *one-hot hashing* - vocabulary is too large to handle explicitly
- ▶ hash words into vectors of fixed size with a very lightweight **hashing function**
- ▶ saves memory and allows online encoding of the data
- ▶ *hash collisions*: two different words may end up with the same hash

Listing 6.4 Using Keras for word-level one-hot encoding

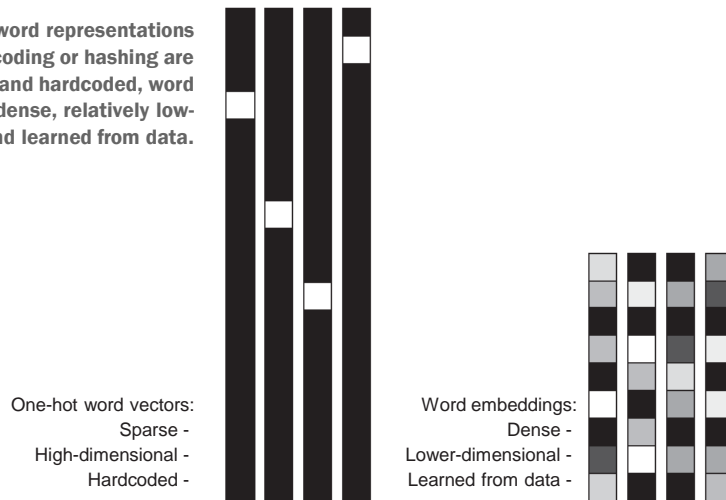
```
samples = ['The cat sat on the mat.', 'The dog ate my homework.']
# 1,000개 이상의 단어가 있다면 hash collisions
dimensionality = 1000
max_length = 10
results = np.zeros((len(samples), max_length, dimensionality))
for i, sample in enumerate(samples):
    for j, word in list(enumerate(sample.split()))[:max_length]:
        # Hashes the word into a random integer index between 0 and 1,000
        index = abs(hash(word)) % dimensionality
        results[i, j, index] = 1.
# in case of dimensionality = 20 → results[0] =
[[0.0.0.0.1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.] The
[0.0.0.1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.] cat
[0.0.0.0.0.0.0.0.0.0.0.0.1.0.0.0.0.0.0.0.0.] sat
[0.0.0.0.0.0.0.0.0.0.0.0.0.1.0.0.0.0.0.0.0.] on
[0.0.0.0.0.0.0.0.0.1.0.0.0.0.0.0.0.0.0.0.0.] the
[0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.1.0.] mat
[0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.]
[0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.]
[0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.]
[0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.]
```

6.1 Working with text data

6.1.2 Using word embeddings

- ▶ **one-hot** encoding are binary, sparse, very high-dimensional (20,000-dimensional or greater)
- ▶ **dense word vectors** also called **word embeddings** - **low-dimensional** floating-point vectors in 256-, 512-, or 1,024-dimensional when dealing with very large vocabularies
- ▶ pack more information into far fewer dimensions

Figure 6.2 Whereas word representations obtained from one-hot encoding or hashing are sparse, high-dimensional, and hardcoded, word embeddings are dense, relatively low-dimensional, and learned from data.



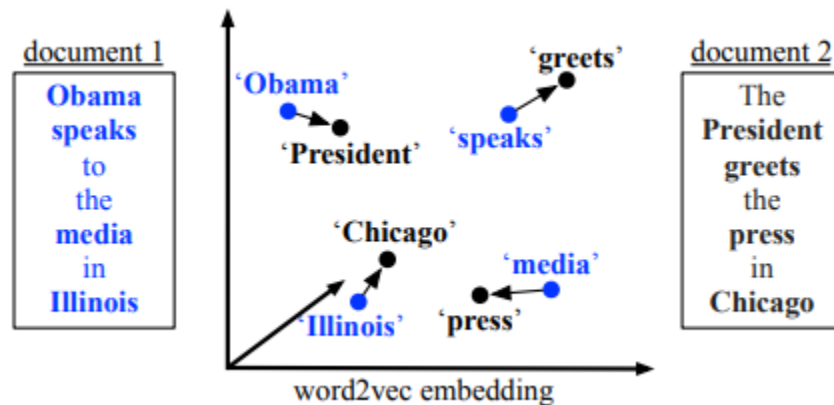
6.1 Working with text data

- ▶ There are two ways to obtain word embeddings:
 - Learn word embeddings jointly with the **main task** you care about (such as **document classification** or **sentiment prediction** → **weights**).
 - *pretrained word embeddings* - Load into your model word embeddings
- ▶ Let's look at both.

6.1 Working with text data

LEARNING WORD EMBEDDINGS WITH THE EMBEDDING LAYER

- ▶ choose the vector at **random** - embedding space has **no structure**: the interchangeable words *accurate* and *exact* end up with completely different embeddings
- ▶ **synonyms** to be embedded into similar word vectors
- ▶ **geometric distance** (such as L2 distance) between any two word vectors to relate to the **semantic distance** between the associated words
- ▶ specific **directions** in the embedding space to be meaningful.



6.1 Working with text data

LEARNING WORD EMBEDDINGS WITH THE EMBEDDING LAYER

- ▶ *cat*, *dog*, *wolf*, and *tiger* - semantic relationships between these words can be encoded as geometric transformations.
- ▶ “from pet to wild animal” - from *cat* to *tiger* and from *dog* to *wolf*
- ▶ “from canine to feline” vector from *dog* to *cat* and from *wolf* to *tiger*
- ▶ “gender” and “plural” vectors - “female” vector + vector “king” → vector “queen,” “plural” vector + vector “king” → “kings.”
- ▶ Word-embedding spaces typically feature thousands of such interpretable and potentially useful vectors.

		Dimensions				
Word vectors	dog	-0.4	0.37	0.02	-0.34	
	cat	-0.15	-0.02	-0.23	-0.23	
	lion	0.19	-0.4	0.35	-0.48	
	tiger	-0.08	0.31	0.56	0.07	
	elephant	-0.04	-0.09	0.11	-0.06	
	cheetah	0.27	-0.28	-0.2	-0.43	
	monkey	-0.02	-0.67	-0.21	-0.48	
	rabbit	-0.04	-0.3	-0.18	-0.47	
	mouse	0.09	-0.46	-0.35	-0.24	
	rat	0.21	-0.48	-0.56	-0.37	

animal
 domesticated
 pet
 fluffy

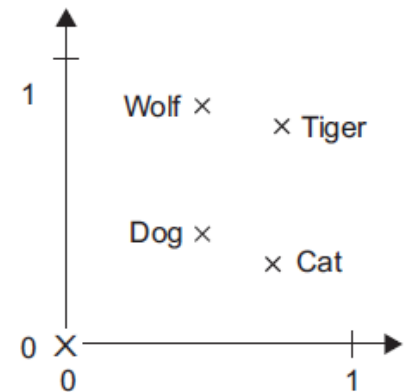


Figure 6.3 A toy example of a word-embedding space

6.1 Working with text data

- ▶ learning the **weights** of a layer: the **Embedding** layer

Listing 6.5 Instantiating an Embedding layer

```
from keras.layers import Embedding  
embedding_layer = Embedding(1000, 64)  
#(batch, input_length)
```

- ▶ The **Embedding** layer is best understood as a dictionary that maps **integer indices** (which stand for specific words) to **dense vectors**.
- ▶ It takes integers as **input**, it looks up these integers in an internal **dictionary**, and it returns the associated **vectors**. It's effectively a dictionary lookup.

Word index → Embedding layer → Corresponding word vector

6.1 Working with text data

- ▶ The Embedding layer takes as input a 2D tensor of integers, of shape `(samples, sequence_length)`, where each entry is a sequence of integers.
- ▶ It can embed sequences of variable lengths: `(32, 10)` (batch of 32 sequences of length 10) or `(64, 15)` (batch of 64 sequences of length 15).
- ▶ All sequences in a batch must have the **same length**, though (because you need to pack them into a single tensor), so sequences that are shorter than others should be **padded with 0s**, and sequences that are longer should be **truncated**.

6.1 Working with text data

- ▶ This layer **returns** a 3D floating-point tensor of shape (samples, sequence_length, **embedding_dimensionality**).
- ▶ Such a 3D tensor can then be processed by an **RNN layer** or a **1D convolution layer** (both will be introduced in the following sections).
- ▶ Embedding layer - its weights (its internal dictionary of token vectors) are initially random → gradually adjusted via backpropagation → embedding space (specialized for the specific problem)
- ▶ IMDB movie-review sentiment-prediction - the **top 10,000** most common words and cut off the reviews after only **20 words**.
- ▶ **input** integer sequences (2D integer tensor) → **embedded sequences** (3D float tensor) → **flatten** the tensor to 2D → **train** a single Dense layer on top for classification → **8-dimensional embeddings** for each of the 10,000 words

6.1 Working with text data

Listing 6.6 Loading the IMDB data for use with an Embedding layer

```
from keras.datasets import imdb
from keras import preprocessing

max_features = 10000 # Number of words to consider as features
maxlen = 20 # Cuts off the text after this number of words
             #(among the max_features most common words)

(x_train, y_train), (x_test, y_test) = imdb.load_data(
    num_words=max_features) # Loads the data as lists of integers

x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen)
# lists of integers → a 2D integer tensor of shape (samples, maxlen)
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)
# padded with 0s for shorter sequences or truncated for longer sequences
```

6.1 Working with text data

Listing 6.7 Using an Embedding layer and classifier on the IMDB data

```
from keras.models import Sequential
from keras.layers import Flatten, Dense

model = Sequential()

model.add(Embedding(10000, 8, input_length=maxlen))
# Specifies the maximum input length to the Embedding layer
# so you can later flatten the embedded inputs.
# Output of the activations have shape (samples, maxlen, 8) of 3D with 8 Output.

model.add(Flatten())
# Flattens the 3D tensor of embeddings into a 2D tensor of shape (samples, maxlen * 8)
model.add(Dense(1, activation='sigmoid')) # Adds the classifier on top
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
model.summary()

history = model.fit(x_train, y_train,
                    epochs=10, batch_size=32, validation_split=0.2)
```

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, 20, 8)	80000
flatten_1 (Flatten)	(None, 160)	0
dense_1 (Dense)	(None, 1)	161

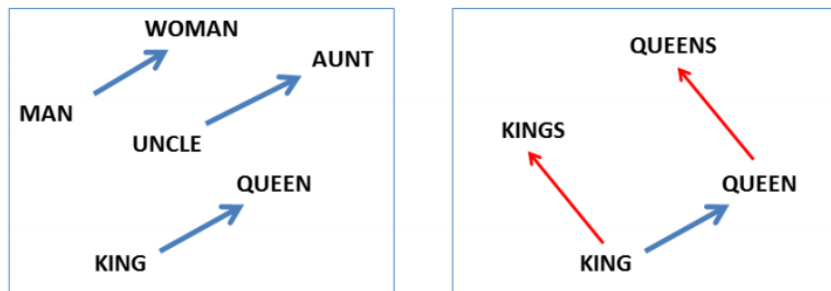
6.1 Working with text data

- ▶ You get to a validation accuracy of ~76%, which is pretty good considering that you're only looking at the first 20 words in every review.
- ▶ no inter-word relationships and sentence structure (for example, this model would likely treat both “this movie is a bomb” and “this movie is the bomb-짱” as being negative reviews).
비슷해 보이지만 a bomb는 폭탄
the bomb는 매우 대단하다 이런 뜻
이걸 잘 반영하지 못했다.는 뜻
- ▶ It's much better to add recurrent layers or 1D convolutional layers on top of the embedded sequences to learn features.

6.1 Working with text data

USING PRETRAINED WORD EMBEDDINGS

- ▶ little training data?
- ▶ precomputed embedding space - **highly structured** and exhibits useful properties by using **word-occurrence statistics**, using a variety of techniques, some involving neural networks.
- ▶ **Word2vec** algorithm (<https://code.google.com/archive/p/word2vec>), developed by Tomas Mikolov at Google in 2013.
 - ▶ Word2vec dimensions capture specific **semantic properties**, such as genders
- ▶ **GloVe**, <https://nlp.stanford.edu/projects/glove>, by Stanford researchers in 2014.
 - ▶ factorizing a matrix of word **co-occurrence statistics** obtained from millions of English tokens, **Wikipedia** data and **Common Crawl** data.



(Mikolov et al., NAACL HLT, 2013)

Window based co-occurrence matrix

- Example corpus:

- I like deep learning.
- I like NLP.
- I enjoy flying.

counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

6.1 Working with text data

6.1.3 Putting it all together: from raw text to word embeddings

- ▶ pretrained word embeddings
- ▶ the original text data instead of using the pretokenized IMDB data packaged in Keras

케라스에서 주는 IMDB가 아닌 오리지널 IMDB를 사용할 것

DOWNLOADING THE IMDB DATA AS RAW TEXT

- ▶ download the raw IMDB dataset from <http://mng.bz/0tIo>.
- ▶ Uncompress it.
- ▶ collect the individual training reviews into a list of strings, one string per review.
- ▶ collect the review labels (positive/negative) into a labels list.

6.1 Working with text data

Listing 6.8 Processing the labels of the raw IMDB data

```
import os

imdb_dir = '/Users/fchollet/Downloads/aclImdb'
train_dir = os.path.join(imdb_dir, 'train')

labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(train_dir, label_type)
    for fname in os.listdir(dir_name):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname))
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)
```

6.1 Working with text data

TOKENIZING THE DATA

▶ **pretrained word embeddings** - restricting the training data to the first 200 samples (otherwise, task-specific embeddings are likely to outperform)

Listing 6.9 Tokenizing the text of the raw IMDB data

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np

maxlen = 100 # Cuts off reviews after 100 words
training_samples = 200 # Trains on 200 samples
validation_samples = 10000 # Validates on 10,000 samples
max_words = 10000 # Considers only the top 10,000 words in the dataset
tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts) # 입력에 맞게 내부 word_index 생성
sequences = tokenizer.texts_to_sequences(texts)
word_index = tokenizer.word_index # {}
print('Found %s unique tokens.' % len(word_index))
data = pad_sequences(sequences, maxlen=maxlen)
labels = np.asarray(labels)
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)
```

```
Found 88582 unique tokens
Shape of data tensor:(25000,100)
Shape of label tensor :(25000,)
```


6.1 Working with text data

```
indices = np.arange(data.shape[0]) # 25,000 - [0 1 2 ... 24997 24998 24999]
# first shuffles the data, all negative first, then all positive
np.random.shuffle(indices) # [23739 2813 974 ... 167 23722 19124]
data = data[indices]
labels = labels[indices]
x_train = data[:training_samples] # 200
y_train = labels[:training_samples] # 200
x_val = data[training_samples:
             training_samples + validation_samples] # 10,000
y_val = labels[training_samples:
               training_samples + validation_samples] # 10,000

x_val: (10000, 100)
x_val: [[ 128 1480 413 ... 188 335 543] [ 7 11 6 ... 52 867 97] [ 23 1487 14 ... 2 65 2776]
        ... [ 0 0 0 ... 42 35 615] [ 480 2 327 ... 39 568 3920] [9141 59 1463 ... 128 232 4572]]
y_val: (10000,)
y_val: [0 1 1 ... 1 0 1]
```

6.1 Working with text data

DOWNLOADING THE GLOVE WORD EMBEDDINGS

- ▶ Go to <https://nlp.stanford.edu/projects/glove>, and download the precomputed embeddings from 2014 English Wikipedia. It's an 822 MB zip file called glove.6B.zip, containing **100-dimensional** embedding vectors for **400,000 words** (or nonword tokens). Unzip it.
- ▶ Let's parse the unzipped file (a .txt file) to build an index that maps words (as strings) to their vector representation (as number vectors).

Listing 6.10 Parsing the GloVe word-embeddings file

```
glove_dir = '/Users/fchollet/Downloads/glove.6B'
embeddings_index = {} # 400,000 words
f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'))
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))
# Found 400000 word vectors
print(len(embeddings_index.get('cat'))))
# 100
# [ 0.23088  0.28283 ... -0.71493 ]
# Word = sandberger
# Coefs = [ 0.28365   -0.6263   , , , -0.15701]
```

6.1 Working with text data

- ▶ Next, you'll build an embedding matrix that you can load into an **Embedding layer**. It must be a matrix of shape `(max_words, embedding_dim)`, where each entry *i* contains the **embedding_dim-dimensional vector** for the word of index *i* in the reference word index (built during tokenization).
- ▶ Note that index **0** isn't supposed to stand for any **word or token**—it's a placeholder.

Listing 6.11 Preparing the GloVe word-embeddings matrix

```
embedding_dim = 100 # len(embeddings_index.get('cat'))

embedding_matrix = np.zeros((max_words, embedding_dim)) # max_words=10000
for word, i in word_index.items(): # [('the', 1), ('and', 2), ..., ('hued', 88582)]
    if i < max_words:
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector

# Words not found in the embedding index will be all zeros.
# [0. 0. 0. ... 0. 0.][-0.038194 -0.24487001 ... 0.27061999] - (10000, 100)
```

6.1 Working with text data

DEFINING A MODEL

- ▶ Let's parse the unzipped file (a .txt file) to build an index that maps words (as strings) to their vector representation (as number vectors).

Listing 6.12 Model definition

```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim,
                    input_length=maxlen))    # maxlen = 20
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

6.1 Working with text data

LOADING THE GLOVE EMBEDDINGS IN THE MODEL

- ▶ The **Embedding layer** has a single weight matrix: a 2D float matrix where each entry i is the word vector meant to be associated with index i .
- ▶ Load the **GloVe matrix** you prepared into the Embedding layer, the first layer in the model
- ▶ **freeze the Embedding layer** (set its trainable attribute to **False**)

Listing 6.13 Loading pretrained word embeddings

```
model.layers[0].set_weights([embedding_matrix])  
#                               (10000, 100)  
model.layers[0].trainable = False
```

6.1 Working with text data

TRAINING AND EVALUATING THE MODEL

- ▶ Compile and train the model.

Listing 6.14 Training and evaluation into the Embedding

```
model.compile(optimizer='rmsprop',  
              loss='binary_crossentropy', metrics=['acc'])  
history = model.fit(x_train, y_train,  
                    epochs=10, batch_size=32,  
                    validation_data=(x_val, y_val))  
model.save_weights('pre_trained_glove_model.h5')
```

6.1 Working with text data

- Now, plot the model's performance over time (see figures 6.5 and 6.6)

Listing 6.15 Plotting the results layer

```
import matplotlib.pyplot as plt
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy') plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

Figure 6.5 Training and validation loss when using pretrained word embeddings

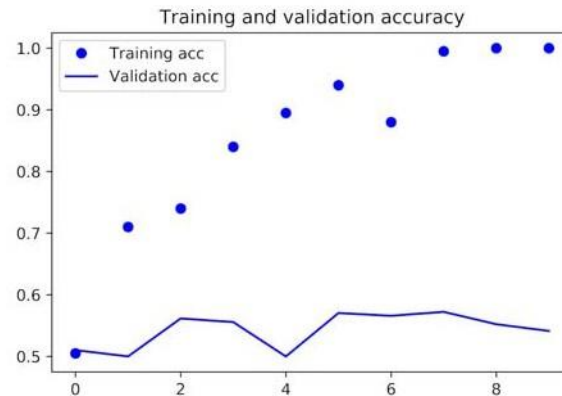
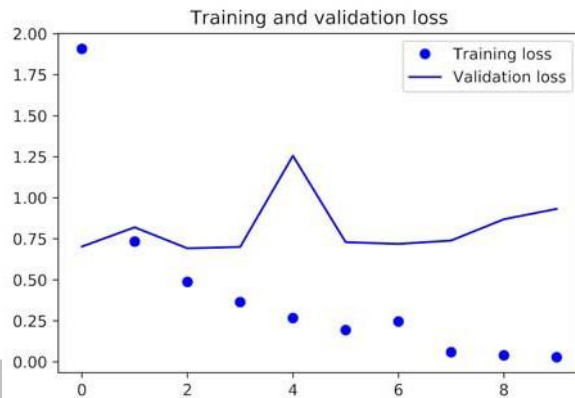


Figure 6.6 Training and validation accuracy when using pretrained word embeddings

6.1 Working with text data

- ▶ The model quickly starts overfitting, which is unsurprising given the small number of training samples. Validation accuracy has high variance for the same reason, but it seems to reach the **high 50s**.
- ▶ so few training samples - performance is heavily dependent on exactly which 200 samples you choose randomly.
- ▶ without loading the pretrained word embeddings and without freezing the embedding layer. In that case, you'll learn a task-specific embedding of the input tokens, which is generally more powerful than pretrained word embeddings when lots of data is available.
- ▶ But in this case, you have only 200 training samples. Let's try it (see figures 6.7 and 6.8).

6.1 Working with text data

Listing 6.16 Training the same model without pretrained word embeddings

```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy', metrics=['acc'])
history = model.fit(x_train, y_train,
                   epochs=10, batch_size=32,
                   validation_data=(x_val, y_val))
```

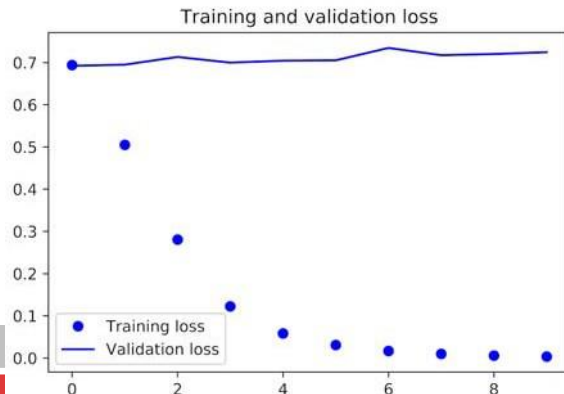
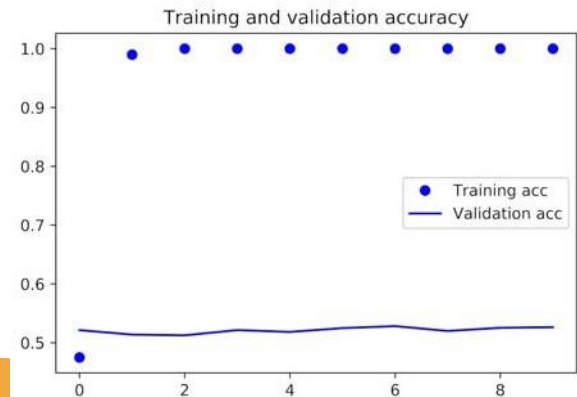


Figure 6.7 Training and validation loss without using pretrained word embeddings

Figure 6.8 Training and validation accuracy without using pretrained word embeddings



6.1 Working with text data

- ▶ Validation accuracy stalls in the **low 50s**. So in this case, pretrained word embeddings outperform jointly learned embeddings. If you **increase the number of training samples**, this will quickly stop being the case—try it as an exercise.
- ▶ Finally, let's evaluate the model on the test data. First, you need to tokenize the test data.

Listing 6.17 Tokenizing the data of the test set

```
test_dir = os.path.join(imdb_dir, 'test')
labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(test_dir, label_type)
    for fname in sorted(os.listdir(dir_name)):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname))
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)

sequences = tokenizer.texts_to_sequences(texts)
x_test = pad_sequences(sequences, maxlen=maxlen)
y_test = np.asarray(labels)
```

6.1 Working with text data

- ▶ Next, load and evaluate the first model

Listing 6.18 Evaluating the model on the test set

```
model.load_weights('pre_trained_glove_model.h5')  
model.evaluate(x_test, y_test)
```

- ▶ You get an appalling test accuracy of **56%**. Working with just a handful of training samples is **difficult!**
 - Turn raw text into something a neural network can process
 - Use the **Embedding layer** in a Keras model to learn task-specific token embeddings
 - Use **pretrained word embeddings** to get an extra boost on small natural-language-processing problems