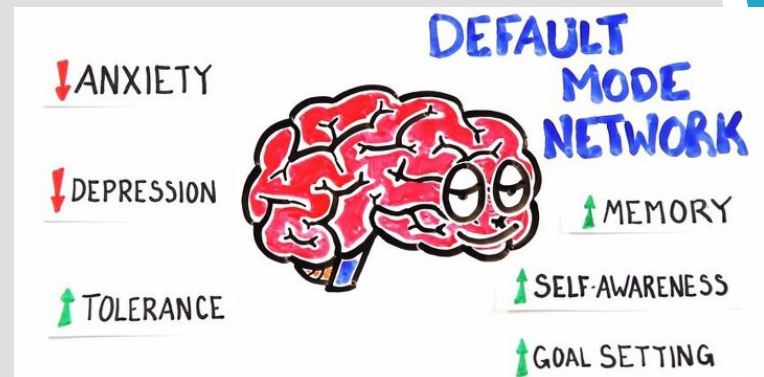
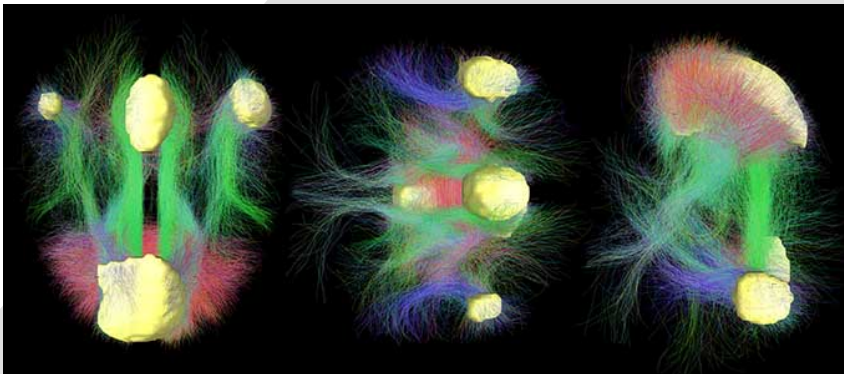


6장 *Deep learning for text and sequences*

“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(topic), 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**

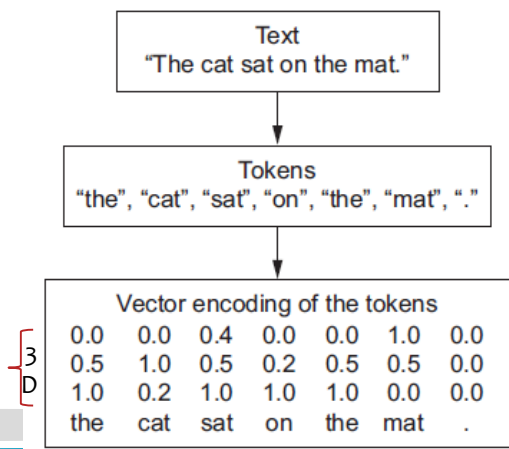


Figure 6.1 From text to tokens to vectors

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 - {key:value}; {'The':1 }
for sample in samples:
    # split() 메서드를 사용해 샘플을 토큰으로 나눕니다.
    for word in sample.split(): # key - word
        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)) # (2,10,11)
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. 1. 0.]	my
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]	the	[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]	homework.
[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.]	

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, 100개
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.
```

token_index = {'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. ... 1. ... 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. ... 1. ... 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 중복 없이 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. 0. 0. 0. 0. 0. ... 0. 0. 0.]
#                    [0. 1. 0. 0. 0. 0. 1. 1. 1. 1. 0. ... 0. 0. 0.]]

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

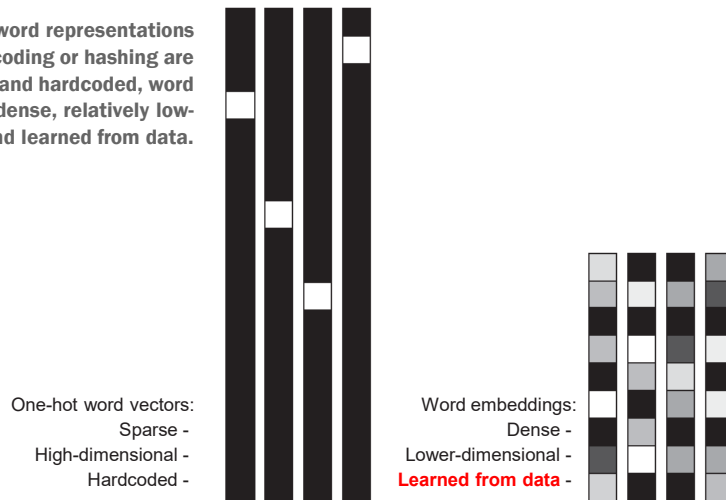
```
samples = ['The cat sat on the mat.', 'The dog ate my homework.']
# 1,000개 이상의 단어가 있다면 hash collisions
dimensionality = 1000
max_length = 10
results = np.zeros((len(samples), max_length, dimensionality))
for i, sample in enumerate(samples):
    for j, word in list(enumerate(sample.split()))[:max_length]:
        # 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] =
[[0.0.0.0.0.1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.] The
[0.0.0.0.1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.] cat
[0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.1.0.0.0.0.0.0.0.] sat
[0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.1.0.0.0.0.0.0.] on
[0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.1.0.] the
[0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.1.0.] 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.]
[0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.]
```

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.



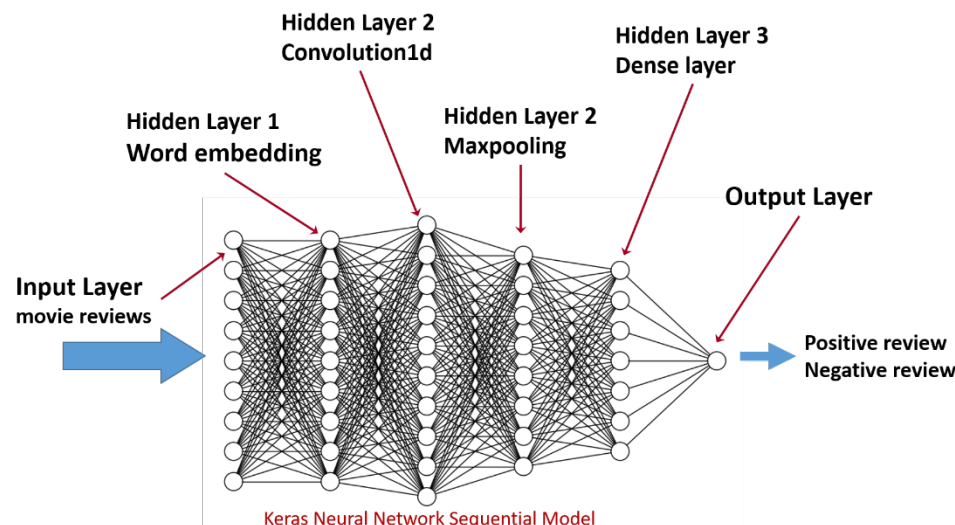
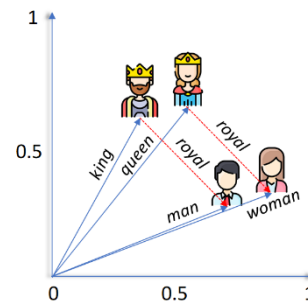
6.1 Working with text data

- ▶ 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** in a neural network).
 - *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



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 - 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

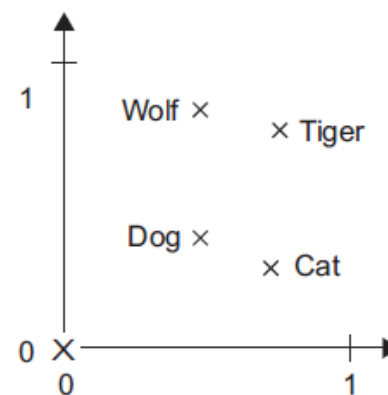


Figure 6.3 A toy example of a word-embedding space

6.1 Working with text data

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

Listing 6.5 Instantiating an Embedding layer

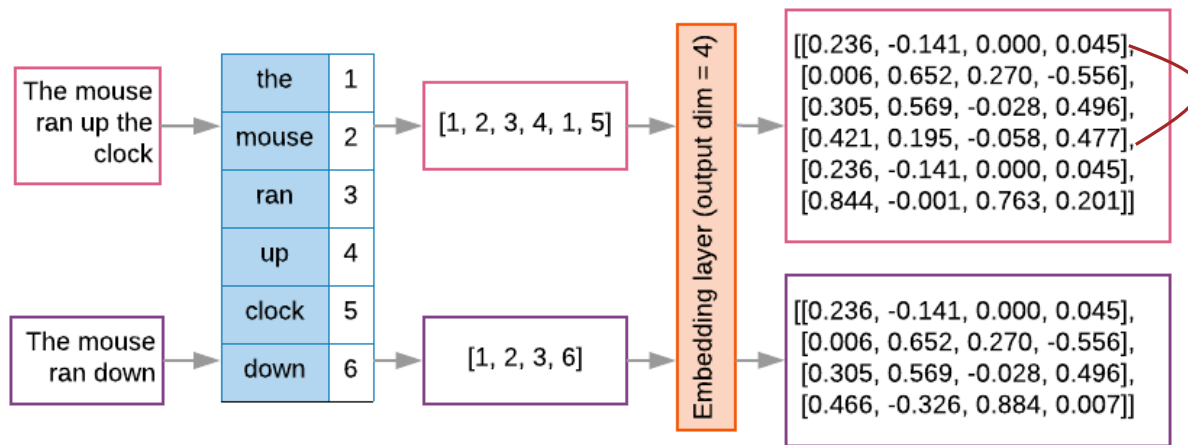
```
from keras.layers import Embedding  
embedding_layer = Embedding(1000, 64)  
#(batch, input_length)
```

- ▶ 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)
# training dataset-8000개, test dataset-2000개
```

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

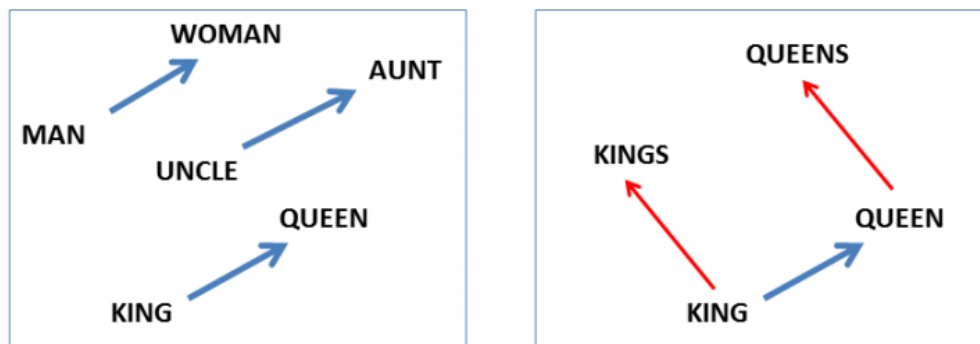
6.1 Working with text data

- ▶ You get to a validation accuracy of $\sim 76\%$, 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).
- ▶ 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



(Mikolov et al., NAACL HLT, 2013)

6.1 Working with text data

USING PRETRAINED WORD EMBEDDINGS

► **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.

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

