텍스트와 시퀀스를 위한 딥러닝

이 장에서 다룰 핵심 내용

- 텍스트 데이터를 유용한 형태로 전처리하는 방법
- 순환 신경망을 사용하는 방법
- 1D 컨브넷을 사용한 시퀀스 데이터의 처리

이 장에서는 텍스트(단어의 시퀀스 또는 문자의 시퀀스), 시계열 또는 일반적인 시퀀스(sequence) 데이터를 처리할 수 있는 딥러닝 모델을 살펴보겠습니다. 시퀀스 데이터를 처리하는 기본적인 딥러닝 모델은 **순환 신경망**(recurrent neural network)과 **1D 컨브넷**(1D convnet) 두 가지입니다. 1D 컨브넷은 이전 장에서 다룬 2D 컨브넷의 1차원 버전입니다. 이 장에서는 두 가지 방법을 모두 다루겠습니다.

다음 애플리케이션들이 이런 알고리즘을 사용합니다.

- 문서 분류나 시계열 분류. 예를 들어 글의 주제나 책의 저자 식별하기
- 시계열 비교. 예를 들어 두 문서나 두 주식 가격이 얼마나 밀접하게 관련이 있는지 추정하기
- 시퀀스-투-시퀀스 학습. 예를 들어 영어 문장을 프랑스어로 변환하기
- 감성 분석. 예를 들어 트윗이나 영화 리뷰가 긍정적인지 부정적인지 분류하기
- 시계열 예측. 예를 들어 어떤 지역의 최근 날씨 데이터가 주어졌을 때 향후 날씨 예측하기

이 장의 예제는 2개의 문제를 집중하여 다룹니다. 앞서 다루어 보았던 IMDB 데이터셋의 감성 분석과 기온 예측입니다. 이 두 작업에서 사용한 기법들은 위에 나열한 것은 물론 많은 애플리케이션들과 관련되어 있습니다.

DEEP LEARNING

6.1 텍스트 데이터 다루기

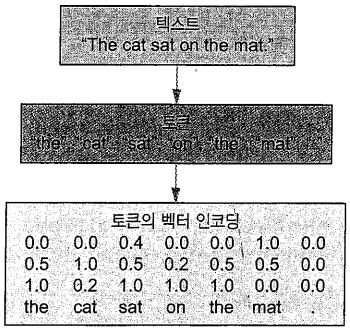
텍스트는 가장 흔한 시퀀스 형태의 데이터입니다. 텍스트는 단어의 시퀀스나 문자의 시퀀스로 이해할 수 있습니다. 보통 단어 수준으로 작업하는 경우가 많습니다. 다음 절에서 소개할 시퀀스 처리용 딥러닝 모델은 텍스트를 사용하여 기초적인 자연어 이해(natural language understanding) 문제를 처리할 수 있습니다. 이런 모델은 문서 분류, 감성 분석, 저자 식별, (제한된 범위의) 질문 응답(Question Answering, QA) 등의 애플리케이션에 적합합니다. 물론 이런 딥러닝 모델이 사람처럼 진짜 텍스트를 이해하는 것은 아닙니다. 이 장을 읽을 때 이런 점을 유념해 주세요. 이런 모델은 문자 언어(written language)에 대한 통계적 구조를 만들어 간단한 텍스트 문제를 해결합니다. 컴퓨터 비전이 픽셀에 적용한 패턴 인식(pattern recognition)인 것처럼 자연어 처리(natural language processing)를 위한 딥러닝은 단어, 문장, 문단에 적용한 패턴 인식입니다.

다른 모든 신경망과 마찬가지로 텍스트 원본을 입력으로 사용하지 못합니다. 딥러닝 모델은 수치형 텐서만 다룰 수 있습니다. 텍스트를 수치형 텐서로 변환하는 과정을 **텍스트 벡터화**(vectorizing text)라고 합니다. 여기에는 여러 가지 방식이 있습니다.

- 텍스트를 단어로 나누고 각 단어를 하나의 벡터로 변환합니다.
- 텍스트를 문자로 나누고 각 문자를 하나의 벡터로 변환합니다.
- 텍스트에서 단어나 문자의 **n-그램**(n-gram)을 추출하여 각 n-그램을 하나의 벡터로 변환합니다. n-그램은 연속된 단어나 문자의 그룹으로 텍스트에서 단어나 문자를 하나씩 이동하면서 추출합니다.

텍스트를 나누는 이런 단위(단어, 문자, n-그램)를 **토큰**(token)이라고 합니다. 그리고 텍스트를 토큰으로 나누는 작업을 **토큰화**(tokenization)라고 합니다. 모든 텍스트 벡터화 과정은 어떤 종류의 토큰화를 적용하고 생성된 토큰에 수치형 벡터를 연결하는 것으로 이루어집니다. 이런 벡터는 시퀀스 텐서로 묶여져서 심층 신경망에 주입됩니다. 토큰과 벡터를 연결하는 방법은 여러 가지가 있습니다. 이 절에서 두 가지 주요 방법을 소개하겠습니다. 토큰의 **원-핫 인코딩**(one-hot encoding)과 **토큰 임베딩**(token embedding)(일반적으로 단어에 대해서만 사용되므로 **단어 임베딩**(word embedding)이라고도 부릅니다)입니다. 이 절에서 이 기법들을 설명하고 이를 이용하여 원본 텍스트를 케라스에서 사용할 수 있도록 넘파이 배열로 바꾸는 방법을 소개하겠습니다.

❯ 그림 6-1 텍스트에서 토큰으로, 토큰에서 벡터로



Note ≡ n-그램과 BoW

단어 n-그램은 문장에서 추출한 N개(또는 그 이하)의 연속된 단어 그룹입니다. 같은 개념이 단어 대신 문자에도 적용될 수 있습니다.

다음은 간단한 예입니다. "The cat sat on the mat."이란 문장을 생각해 보죠. 이 문장은 다음 2-그램의 집합으로 분해할 수 있습니다.

```
{"The", "The cat", "cat", "cat sat", "sat",
  "sat on", "on", "on the", "the", "the mat", "mat"}
```

또 다음 3-그램의 집합으로도 분해할 수 있습니다.

```
{"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"}
```

이런 집합을 각각 2-그램 가방(bag of 2-gram) 또는 3-그램 가방(bag of 3-gram)이라고 합니다. 가방(bag)이란 용어는 다루고자 하는 것이 리스트나 시퀀스가 아니라 토큰의 집합이라는 사실을 의미합니다. 이 토큰에는 특정한 순서가 없습니다. 이런 종류의 토큰화 방법을 BoW(Bag-of-Words)라고 합니다.

BoW가 순서가 없는 토큰화 방법이기 때문에(생성된 토큰은 시퀀스가 아니라 집합으로 간주되고 문장의 일반적인 구조가 사라집니다) 딥러닝 모델보다 얕은 학습 방법의 언어 처리 모델에 사용되는 경향이 있습니다. n-그램을 추출하는 것은 일종의 특성 공학입니다. 딥러닝은 유연하지 못하고 불안정한 이런 방식을 계층적인 특성 학습으로 대체합니다. 나중에 소개할 순환 신경망과 1D 컨브넷으로 단어와 문자 그룹에 대한 특성을 학습할 수 있습니다. 이 방식들은 그룹들을 명시적으로 알려 주지 않아도 연속된 단어나 문자의 시퀀스를 봄으로써 학습합니다. 이런 이유 때문에 이 책에서는 n-그램을 더 다루지 않습니다. 하지만 로지스틱 회귀나 랜덤 포레스트 같은 얕은 학습 방법의 텍스트 처리 모델을 사용할 때는 강력하고 아주 유용한 특성 공학 방법임을 기억해 두세요.[1]

6.1.1 단어와 문자의 원-핫 인코딩

원-핫 인코딩은 토큰을 벡터로 변환하는 가장 일반적이고 기본적인 방법입니다. 3장에서 IMDB와 로이터 예제에서 이를 보았습니다(단어의 원-핫 인코딩을 사용했습니다). 모든 단어에 고유한 정수 인덱스를 부여하고 이 정수 인덱스 i를 크기가 N(어휘 사전의 크기)인 이진 벡터로 변환합니다. 이 벡터는 i번째 원소만 1이고 나머지는 모두 0입니다.

1 옮주 BoW는 각 샘플을 어휘 사전 크기의 벡터로 변환합니다. 이 벡터의 원소는 특정 토큰 하나에 대응하며 순서가 없습니다. 변환은 샘플에 나타난 토큰의 인덱스 위치를 1로 바꾸는 식입니다. 전체 데이터셋의 크기는 (samples, vocabulary_size)가 됩니다. BoW와 n-그램을 사용한 텍스트 분석은 〈파이썬 라이브러리를 활용한 머신러닝〉(한빛미디어, 2017)의 7장을 참고하세요.

물론 원-핫 인코딩은 문자 수준에서도 적용할 수 있습니다. 원-핫 인코딩이 무엇이고 어떻게 구현하는지 명확하게 설명하기 위해 코드 6-1과 코드 6-2에 단어와 문자에 대한 간단한 예를 만들었습니다.

코드 6-1 단어 수준의 원-핫 인코딩하기(간단한 예)

```
import numpy as np

samples = ['The cat sat on the mat.', 'The dog ate my homework.']
          ┊------ 초기 데이터: 각 원소가 샘플입니다. (이 예에서 하나의 샘플이 하나의 문장입니다. 하지만 문서 전체가 될 수도 있습니다.)
token_index = {} -------- 데이터에 있는 모든 토큰의 인덱스를 구축합니다.
for sample in samples:
    for word in sample.split():
                  ┊------- split() 메서드를 사용하여 샘플을 토큰으로 나눕니다. 실전에서는 구두점과 특수 문자도 사용합니다.
        if word not in token_index:
            token_index[word] = len(token_index) + 1
                                ┊------- 단어마다 고유한 인덱스를 할당합니다. 인덱스 0은 사용하지 않습니다.[2]
max_length = 10 -------- 샘플을 벡터로 변환합니다. 각 샘플에서 max_length까지 단어만 사용합니다.

results = np.zeros(shape=(len(samples),
                          max_length,
                          max(token_index.values()) + 1)) -------- 결과를 저장할 배열입니다.
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.
```

코드 6-2 문자 수준 원-핫 인코딩하기(간단한 예)

```
import string

samples = ['The cat sat on the mat.', 'The dog ate my homework.']
characters = string.printable -------- 출력 가능한 모든 아스키(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):
```

2 옮긴 관례상 인덱스 0은 단어가 아닌 토큰을 위해 남겨 둡니다. 예를 들어 코드 6-6에서 시퀀스를 패딩할 때 0을 사용하여 채웁니다. 코드 6-3에서 Tokenizer(num_words=1000)처럼 지정하면 인덱스 0을 제외하고 최대 999개까지 단어를 사용합니다. num_words 매개변수를 지정하지 않으면 고유한 단어의 전체 개수 + 1(인덱스 0)이 원-핫 인코딩의 길이가 됩니다.

```
    for j, character in enumerate(sample):
        index = token_index.get(character)
        results[i, j, index] = 1.
```

케라스에는 원본 텍스트 데이터를 단어 또는 문자 수준의 원-핫 인코딩으로 변환해 주는 유틸리티가 있습니다. 특수 문자를 제거하거나 빈도가 높은 N개의 단어만 선택(입력 벡터 공간이 너무 커지지 않도록 하기 위한 일반적인 제한 방법입니다)하는 등 여러 가지 중요한 기능들이 있기 때문에 이 유틸리티를 사용하는 것이 좋습니다.[3]

코드 6-3 케라스를 사용한 단어 수준의 원-핫 인코딩하기

```
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)
tokenizer.fit_on_texts(samples) -------- 단어 인덱스를 구축합니다.

sequences = tokenizer.texts_to_sequences(samples) -------- 문자열을 정수 인덱스의 리스트로 변환합니다.

one_hot_results = tokenizer.texts_to_matrix(samples, mode='binary')
                  ------- 직접 원-핫 이진 벡터 표현을 얻을 수 있습니다. 원-핫 인코딩 외에 다른 벡터화 방법들도 제공합니다.[4]
word_index = tokenizer.word_index -------- 계산된 단어 인덱스를 구합니다.
print('%s개의 고유한 토큰을 찾았습니다.' % len(word_index))
```

원-핫 인코딩의 변종 중 하나는 **원-핫 해싱**(one-hot hashing) 기법입니다. 이 방식은 어휘 사전에 있는 고유한 토큰의 수가 너무 커서 모두 다루기 어려울 때 사용합니다. 각 단어에 명시적으로 인덱스를 할당하고 이 인덱스를 딕셔너리에 저장하는 대신에 단어를 해싱하여 고정된 크기의 벡터로 변환합니다. 일반적으로 간단한 해싱 함수를 사용합니다. 이 방식의 주요 장점은 명시적인 단

3 역주 코드 6-1과 코드 6-2의 results 배열의 크기는 (samples, max_length, token_length + 1)이고 코드 6-3의 one_hot_results 크기는 (samples, max_length)입니다. 케라스의 Tokenizer 클래스가 만드는 원-핫 인코딩은 1-그램의 BoW와 같습니다.

4 역주 texts_to_matrix() 메서드는 텍스트를 시퀀스 리스트로 바꾸어 주는 texts_to_sequences() 메서드와 시퀀스 리스트를 넘파이 배열로 바꾸어 주는 sequences_to_matrix() 메서드를 차례대로 호출합니다. mode 매개변수에서 지원하는 값은 기본값 'binary' 외에 'count', 'freq', 'tfidf'가 있습니다. 'count'는 단어의 출현 횟수를 사용하고 'freq'는 출현 횟수를 전체 시퀀스의 길이로 나누어 정규화합니다. 'tfidf'는 TF-IDF 방식(https://bit.ly/2tNbg7G)을 의미합니다.

어 인덱스[5]가 필요 없기 때문에 메모리를 절약하고 온라인 방식으로 데이터를 인코딩할 수 있습니다(전체 데이터를 확인하지 않고 토큰을 생성할 수 있습니다). 한 가지 단점은 **해시 충돌**(hash collision)입니다. 2개의 단어가 같은 해시를 만들면 이를 바라보는 머신 러닝 모델은 단어 사이의 차이를 인식하지 못합니다. 해싱 공간의 차원이 해싱될 고유 토큰의 전체 개수보다 훨씬 크면 해시 충돌의 가능성은 감소합니다.

코드 6-4 해싱 기법을 사용한 단어 수준의 원-핫 인코딩하기(간단한 예)

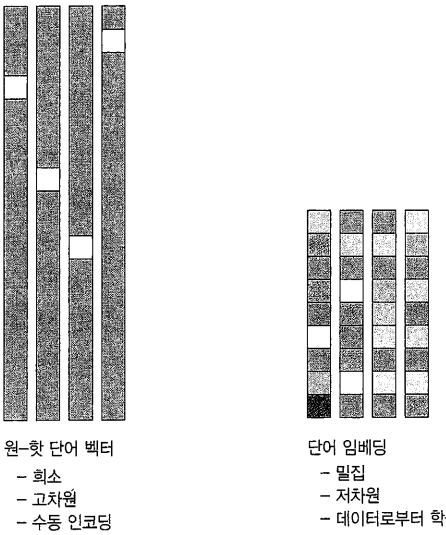
```
samples = ['The cat sat on the mat.', 'The dog ate my homework.']

dimensionality = 1000 ------------ 단어를 크기가 1,000인 벡터로 저장합니다. 1,000개(또는 그 이상)의 단어가
max_length = 10                    있다면 해싱 충돌이 늘어나고 인코딩의 정확도가 감소될 것입니다.

results = np.zeros((len(samples), max_length, dimensionality))
for i, sample in enumerate(samples):
    for j, word in list(enumerate(sample.split()))[:max_length]:
        index = abs(hash(word)) % dimensionality -------- 단어를 해싱하여 0과 1,000 사이의 랜덤한
        results[i, j, index] = 1.                         정수 인덱스로 변환합니다.
```


6.1.2 단어 임베딩 사용하기

단어와 벡터를 연관 짓는 강력하고 인기 있는 또 다른 방법은 **단어 임베딩**이라는 밀집 **단어 벡터**(word vector)를 사용하는 것입니다. 원-핫 인코딩으로 만든 벡터는 희소(sparse)하고(대부분 0으로 채워집니다) 고차원입니다(어휘 사전에 있는 단어의 수와 차원이 같습니다). 반면에 단어 임베딩은 저차원의 실수형 벡터입니다(희소 벡터의 반대인 밀집 벡터입니다). 그림 6-2를 참고하세요. 원-핫 인코딩으로 얻은 단어 벡터와 달리 단어 임베딩은 데이터로부터 학습됩니다. 보통 256차원, 512차원 또는 큰 어휘 사전을 다룰 때는 1,024차원의 단어 임베딩을 사용합니다. 반면에 원-핫 인코딩은 (2만 개의 토큰으로 이루어진 어휘 사전을 만들려면) 20,000차원 또는 그 이상의 벡터일 경우가 많습니다. 따라서 단어 임베딩이 더 많은 정보를 적은 차원에 저장합니다.

5 역주 코드 6-3의 word_index를 말합니다. 이 딕셔너리의 키(key)는 단어고 값(value)은 단어의 인덱스입니다.

❮ 그림 6-2 원-핫 인코딩이나 해싱으로 얻은 단어 표현은 희소하고 고차원이고 수동으로 인코딩되지만, 단어 임베딩은 조밀하고 비교적 저차원이며 데이터로부터 학습된다



단어 임베딩을 만드는 방법은 두 가지입니다.

- (문서 분류나 감성 예측 같은) 관심 대상인 문제와 함께 단어 임베딩을 학습합니다. 이런 경우에는 랜덤한 단어 벡터로 시작해서 신경망의 가중치를 학습하는 것과 같은 방식으로 단어 벡터를 학습합니다.
- 풀려는 문제가 아니고 다른 머신 러닝 작업에서 미리 계산된 단어 임베딩을 로드합니다. 이를 **사전 훈련된 단어 임베딩**(pretrained word embedding)이라고 합니다.

두 가지 모두 살펴보겠습니다.

Embedding 층을 사용하여 단어 임베딩 학습하기

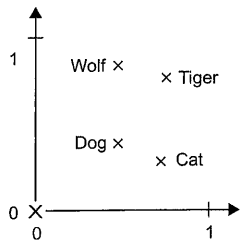
단어와 밀집 벡터를 연관 짓는 가장 간단한 방법은 랜덤하게 벡터를 선택하는 것입니다. 이 방식의 문제점은 임베딩 공간이 구조적이지 않다는 것입니다. 예를 들어 accurate와 exact 단어는 대부분 문장에서 비슷한 의미로 사용되지만 완전히 다른 임베딩을 가집니다. 심층 신경망이 이런 임의의 구조적이지 않은 임베딩 공간을 이해하기는 어렵습니다.

단어 벡터 사이에 좀 더 추상적이고 기하학적인 관계를 얻으려면 단어 사이에 있는 의미 관계를 반영해야 합니다. 단어 임베딩은 언어를 기하학적 공간에 매핑하는 것입니다. 예를 들어 잘 구축

된 임베딩 공간에서는 동의어가 비슷한 단어 벡터로 임베딩될 것입니다. 일반적으로 두 단어 벡터 사이의 거리(L2 거리)는 이 단어 사이의 의미 거리와 관계되어 있습니다(멀리 떨어진 위치에 임베딩된 단어 의미는 서로 다른 반면에 비슷한 단어들은 가까이 임베딩됩니다). 거리 외에 임베딩 공간의 특정 방향도 의미를 가질 수 있습니다. 확실한 이해를 위해 구체적인 예를 살펴보겠습니다.

그림 6-3에서 4개의 단어 cat, dog, wolf, tiger가 2D 평면에 임베딩되어 있습니다. 이 벡터 표현을 사용하여 단어 간의 의미 관계를 기하학적 변환으로 인코딩할 수 있습니다. 예를 들어 cat에서 tiger로 이동하는 것과 dog에서 wolf로 이동하는 것을 같은 벡터로 나타낼 수 있습니다. 이 벡터는 '애완동물에서 야생 동물로 이동'하는 것으로 해석할 수 있습니다. 비슷하게 다른 벡터로 dog에서 cat으로 이동하는 것과 wolf에서 tiger로 이동하는 것을 나타내면 '개과에서 고양이과로 이동'하는 벡터로 해석할 수 있습니다.

❤ 그림 6-3 단어 임베딩 공간의 간단한 예



실제 단어 임베딩 공간에서 의미 있는 기하학적 변환의 일반적인 예는 '성별' 벡터와 '복수(plural)' 벡터입니다. 예를 들어 'king' 벡터에 'female' 벡터를 더하면 'queen' 벡터가 됩니다. 'plural' 벡터를 더하면 'kings'가 됩니다. 단어 임베딩 공간은 전형적으로 이런 해석 가능하고 잠재적으로 유용한 수천 개의 벡터를 특성으로 가집니다.

사람의 언어를 완벽하게 매핑해서 어떤 자연어 처리 작업에도 사용할 수 있는 이상적인 단어 임베딩 공간이 있을까요? 아마도 가능하겠지만 아직까지 이런 종류의 공간은 만들지 못했습니다. 사람의 언어에도 그런 것은 없습니다. 세상에는 많은 다른 언어가 있고 언어는 특정 문화와 환경을 반영하기 때문에 서로 동일하지 않습니다. 실제로 좋은 단어 임베딩 공간을 만드는 것은 문제에 따라 크게 달라집니다. 영어로 된 영화 리뷰 감성 분석 모델을 위한 완벽한 단어 임베딩 공간은 영어로 된 법률 문서 분류 모델을 위한 완벽한 임베딩 공간과 다를 것 같습니다. 특정 의미 관계의 중요성이 작업에 따라 다르기 때문입니다.

따라서 새로운 작업에는 새로운 임베딩을 학습하는 것이 타당합니다. 다행히 역전파를 사용하여 쉽게 만들 수 있고 케라스를 사용하면 더 쉽습니다. `Embedding` 층의 가중치를 학습하면 됩니다.

코드 6-5 Embedding 층의 객체 생성하기

```
from keras.layers import Embedding

embedding_layer = Embedding(1000, 64)
```

Embedding 층은 적어도 2개의 매개변수를 받습니다. 가능한 토큰의 개수(여기서는 1,000으로 단어 인덱스 최댓값 + 1입니다)와 임베딩 차원(여기서는 64)입니다.[6]

Embedding 층을 (특정 단어를 나타내는) 정수 인덱스를 밀집 벡터로 매핑하는 딕셔너리로 이해하는 것이 가장 좋습니다. 정수를 입력으로 받아 내부 딕셔너리에서 이 정수에 연관된 벡터를 찾아 반환합니다. 딕셔너리 탐색은 효율적으로 수행됩니다[7](그림 6-4 참고).

❤ **그림 6-4** Embedding 층

Embedding 층은 크기가 (samples, sequence_length)인 2D 정수 텐서를 입력으로 받습니다. 각 샘플은 정수의 시퀀스입니다. 가변 길이의 시퀀스를 임베딩할 수 있습니다. 예를 들어 앞 예제의 Embedding 층에 (32, 10) 크기의 배치(길이가 10인 시퀀스 32개로 이루어진 배치)나 (64, 15) 크기의 배치(길이가 15인 시퀀스 64개로 이루어진 배치)를 주입할 수 있습니다. 배치에 있는 모든 시퀀스는 길이가 같아야 하므로(하나의 텐서에 담아야 하기 때문에) 작은 길이의 시퀀스는 0으로 패딩되고 길이가 더 긴 시퀀스는 잘립니다.

Embedding 층은 크기가 (samples, sequence_length, embedding_dimensionality)인 3D 실수형 텐서를 반환합니다. 이런 3D 텐서는 RNN 층이나 1D 합성곱 층에서 처리됩니다(둘 다 이어지는 절에서 소개하겠습니다).

Embedding 층의 객체를 생성할 때 가중치(토큰 벡터를 위한 내부 딕셔너리)는 다른 층과 마찬가지로 랜덤하게 초기화됩니다. 훈련하면서 이 단어 벡터는 역전파를 통해 점차 조정되어 이어지는 모델이 사용할 수 있도록 임베딩 공간을 구성합니다. 훈련이 끝나면 임베딩 공간은 특정 문제에 특화된 구조를 많이 가지게 됩니다.

이를 익숙한 IMDB 영화 리뷰 감성 예측 문제에 적용해 보죠. 먼저 데이터를 준비합니다. 영화 리뷰에서 가장 빈도가 높은 1만 개의 단어를 추출하고(처음 이 데이터셋으로 작업했던 것과 동일합니다) 리뷰에서 20개가 넘는 단어는 버립니다. 이 네트워크는 1만 개의 단어에 대해 8차원의 임베

6 옮긴 이 Embedding 층의 가중치 크기는 (1000, 64)입니다. 인덱스 0은 사용하지 않으므로 단어 인덱스는 1~999 사이입니다.

7 옮긴 텐서플로 백엔드를 사용할 경우 tf.nn.embedding_lookup() 함수를 사용하여 병렬 처리됩니다.

딩을 학습하여 정수 시퀀스 입력(2D 정수 텐서)을 임베딩 시퀀스(3D 실수형 텐서)로 바꿀 것입니다. 그다음 이 텐서를 2D로 펼쳐서 분류를 위한 Dense 층을 훈련하겠습니다.

코드 6-6 Embedding 층에 사용할 IMDB 데이터 로드하기

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

max_features = 10000 -------- 특성으로 사용할 단어의 수
maxlen = 20 -------- 사용할 텍스트의 길이(가장 빈번한 max_features개의 단어만 사용합니다.)

(x_train, y_train), (x_test, y_test) = imdb.load_data(
    num_words=max_features) -------- 정수 리스트로 데이터를 로드합니다.

                                  -------- 리스트를 (samples, maxlen) 크기의 2D 정수 텐서로 변환합니다.[8]
x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)
```

코드 6-7 IMDB 데이터에 Embedding 층과 분류기 사용하기

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

model = Sequential()
model.add(Embedding(10000, 8, input_length=maxlen)) ------ 나중에 임베딩된 입력을 Flatten 층에서 펼치기 위해 Embedding 층에 input_length를 지정합니다.[9] Embedding 층의 출력 크기는 (samples, maxlen, 8)이 됩니다.

model.add(Flatten()) -------- 3D 임베딩 텐서를 (samples, maxlen * 8) 크기의 2D 텐서로 펼칩니다.

model.add(Dense(1, activation='sigmoid')) -------- 분류기를 추가합니다.
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)
```

8 역주 pad_sequences() 함수에는 패딩을 넣을 위치를 지정하는 padding 매개변수가 있습니다. 기본값 'pre'는 시퀀스 왼쪽에 패딩이 추가되고 'post'는 오른쪽에 패딩이 추가됩니다. 이 예에서는 문장에서 20개의 단어만 사용하므로 실제 패딩되는 샘플은 없습니다.

9 역주 Embedding 층에 input_length를 지정하지 않으면 출력 텐서의 차원이 (None, None, 8)이 됩니다. Flatten 층을 사용하려면 첫 번째 배치 차원을 제외하고 나머지 차원이 결정되어야 하므로 input_length를 지정하여 (None, 20, 8)로 만듭니다.

약 75% 정도의 검증 정확도가 나옵니다. 리뷰에서 20개의 단어만 사용한 것치고 꽤 좋은 결과입니다. 하지만 임베딩 시퀀스를 펼치고 하나의 Dense 층을 훈련했으므로 입력 시퀀스에 있는 각 단어를 독립적으로 다루었습니다. 단어 사이의 관계나 문장 구조를 고려하지 않았습니다(예를 들어 이 모델은 "this movie is a bomb"과 "this movie is the bomb"을 부정적인 리뷰로 동일하게 다룰 것입니다[10]). 각 시퀀스 전체를 고려한 특성을 학습하도록 임베딩 층 위에 순환 층이나 1D 합성곱 층을 추가하는 것이 좋습니다. 다음 절에서 이에 관해 집중적으로 다루겠습니다.

사전 훈련된 단어 임베딩 사용하기

이따금 훈련 데이터가 부족하면 작업에 맞는 단어 임베딩을 학습할 수 없습니다. 이때는 어떻게 해야 할까요?

풀려는 문제와 함께 단어 임베딩을 학습하는 대신에 미리 계산된 임베딩 공간에서 임베딩 벡터를 로드할 수 있습니다. 이런 임베딩 공간은 뛰어난 구조와 유용한 성질을 가지고 있어서 언어 구조의 일반적인 측면을 잡아낼 수 있습니다. 자연어 처리에서 사전 훈련된 단어 임베딩을 사용하는 이유는 이미지 분류 문제에서 사전 훈련된 컨브넷을 사용하는 이유와 거의 동일합니다. 충분한 데이터가 없어서 자신만의 좋은 특성을 학습하지 못하지만 꽤 일반적인 특성이 필요할 때입니다. 이런 경우에는 다른 문제에서 학습한 특성을 재사용하는 것이 합리적입니다.

단어 임베딩은 일반적으로 (문장이나 문서에 같이 등장하는 단어를 관찰하는) 단어 출현 통계를 사용하여 계산됩니다. 여기에는 여러 가지 기법이 사용되는데 신경망을 사용하는 것도 있고 그렇지 않은 방법도 있습니다. 단어를 위해 밀집된 저차원 임베딩 공간을 비지도 학습 방법으로 계산하는 아이디어는 요슈아 벤지오 등이 2000년대 초에 조사했습니다.[11] 연구나 산업 애플리케이션에 적용되기 시작된 것은 Word2vec 알고리즘(https://code.google.com/archive/p/word2vec)이 등장한 이후입니다. 이 알고리즘은 2013년 구글의 토마스 미코로프(Tomas Mikolov)가 개발했으며, 가장 유명하고 성공적인 단어 임베딩 방법입니다. Word2vec의 차원은 성별처럼 구체적인 의미가 있는 속성을 잡아냅니다.

케라스의 Embedding 층을 위해 내려받을 수 있는 미리 계산된 단어 임베딩 데이터베이스가 여럿 있습니다. Word2vec은 그중 하나입니다. 인기 있는 또 다른 하나는 2014년 스탠포드 대학의 연구자들이 개발한 GloVe(Global Vectors for Word Representation)(https://nlp.stanford.edu/

10 옮주 "this movie is the bomb"은 영화가 아주 좋다는 긍정의 뜻입니다.

11 Yoshua Bengio et al., Neural Probabilistic Language Models (Springer, 2003).

projects/glove)입니다. 이 임베딩 기법은 단어의 동시 출현(co-occurrence)[12] 통계를 기록한 행렬을 분해하는 기법을 사용합니다. 이 개발자들은 위키피디아(Wikipedia) 데이터와 커먼 크롤(Common Crawl) 데이터에서 가져온 수백만 개의 영어 토큰에 대해서 임베딩을 미리 계산해 놓았습니다.

GloVe 임베딩을 케라스 모델에 어떻게 사용하는지 알아보죠. Word2vec 임베딩이나 다른 단어 임베딩 데이터베이스도 방법은 같습니다.[13] 앞서 보았던 텍스트 토큰화 기법도 다시 살펴보겠습니다. 원본 텍스트에서 시작해서 완전한 모델을 구성해 보겠습니다.

6.1.3 모든 내용을 적용하기: 원본 텍스트에서 단어 임베딩까지

앞서 만들었던 것과 비슷한 모델을 사용하겠습니다. 문장들을 벡터의 시퀀스로 임베딩하고 펼친 후 그 위에 Dense 층을 훈련합니다. 여기서는 사전 훈련된 단어 임베딩을 사용하겠습니다. 케라스에 포함된 IMDB 데이터는 미리 토큰화가 되어 있습니다. 이를 사용하는 대신 원본 텍스트 데이터를 내려받아 처음부터 시작하겠습니다.

원본 IMDB 텍스트 내려받기

먼저 http://mng.bz/0tIo에서 IMDB 원본 데이터셋을 내려받고 압축을 해제합니다.[14]

훈련용 리뷰 하나를 문자열 하나로 만들어 훈련 데이터를 문자열의 리스트로 구성해 보죠. 리뷰 레이블(긍정/부정)도 labels 리스트로 만들겠습니다.

코드 6-8 IMDB 원본 데이터 전처리하기

```
import os

imdb_dir = './datasets/aclImdb'
train_dir = os.path.join(imdb_dir, 'train')

labels = []
```

12 역주 언어학에서 형태소나 음소가 올바른 문법의 문장 안에 동시에 나타나는 것을 공기(共起, co-occurrence)라고 합니다(출처: 표준국어대사전). 여기에서는 이해하기 쉬운 표현으로 옮겼습니다.

13 역주 Word2vec을 비롯하여 사전 훈련된 다양한 단어 임베딩은 다음 주소를 참고하세요.
https://bit.ly/2KAz06c

14 역주 번역서의 깃허브는 datasets/aclImdb 폴더에 압축 해제한 IMDB 데이터셋을 포함하고 있기 때문에 별도로 내려받지 않아도 됩니다.

```
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), encoding="utf8")
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)
```

데이터 토큰화

이전 절에서 소개한 개념을 사용하여 텍스트를 벡터로 만들고 훈련 세트와 검증 세트로 나누겠습니다. 사전 훈련된 단어 임베딩은 훈련 데이터가 부족한 문제에 특히 유용합니다(그렇지 않으면 문제에 특화된 임베딩이 훨씬 성능이 좋습니다). 그래서 다음과 같이 훈련 데이터를 처음 200개의 샘플로 제한합니다. 이 모델은 200개의 샘플을 학습한 후 영화 리뷰를 분류할 것입니다.

코드 6-9 IMDB 원본 데이터의 텍스트를 토큰화하기

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

maxlen = 100 -------- 100개 단어 이후는 버립니다.
training_samples = 200 -------- 훈련 샘플은 200개입니다.
validation_samples = 10000 -------- 검증 샘플은 1만 개입니다.
max_words = 10000 -------- 데이터셋에서 가장 빈도 높은 1만 개의 단어만 사용합니다.

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('%s개의 고유한 토큰을 찾았습니다.' % len(word_index))

data = pad_sequences(sequences, maxlen=maxlen)
```

```
labels = np.asarray(labels)
print('데이터 텐서의 크기:', data.shape)
print('레이블 텐서의 크기:', labels.shape)

indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]

x_train = data[:training_samples]
y_train = labels[:training_samples]
x_val = data[training_samples: training_samples + validation_samples]
y_val = labels[training_samples: training_samples + validation_samples]
```

데이터를 훈련 세트와 검증 세트로 분할합니다. 샘플이 순서대로 있기 때문에 (부정 샘플이 모두 나온 후 긍정 샘플이 옵니다) 먼저 데이터를 섞습니다.

GloVe 단어 임베딩 내려받기

https://nlp.stanford.edu/projects/glove에서 2014년 영문 위키피디아를 사용하여 사전에 계산된 임베딩을 내려받습니다. 이 파일의 이름은 glove.6B.zip이고 압축 파일 크기는 823MB입니다. 40만 개의 단어(또는 단어가 아닌 토큰)에 대한 100차원의 임베딩 벡터를 포함하고 있습니다.[15] datasets 폴더 아래에 파일의 압축을 풉니다.

임베딩 전처리

압축 해제한 파일(.txt 파일)을 파싱하여 단어(즉 문자열)와 이에 상응하는 벡터 표현(즉 숫자 벡터)을 매핑하는 인덱스를 만듭니다.

코드 6-10 GloVe 단어 임베딩 파일 파싱하기

```
glove_dir = './datasets/'

embeddings_index = {}
f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'), encoding="utf8")
for line in f:
    values = line.split()
    word = values[0]
```

15 역주 glove.6B.zip 파일에는 50차원, 100차원, 200차원, 300차원의 임베딩 벡터 파일이 들어 있습니다. 이 파일은 용량 때문에 깃허브에 올릴 수 없어 따로 내려받아야 합니다. 스탠포드 대학 웹 사이트에 접근이 안 될 경우 다음 주소에서 내려받으세요(https://bit.ly/2NlJwdb).

```
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('%s개의 단어 벡터를 찾았습니다.' % len(embeddings_index))
```

그다음 Embedding 층에 주입할 수 있도록 임베딩 행렬을 만듭니다. 이 행렬의 크기는 (max_words, embedding_dim)이어야 합니다. 이 행렬의 i번째 원소는 (토큰화로 만든) 단어 인덱스의 i번째 단어에 상응하는 embedding_dim차원 벡터입니다. 인덱스 0은 어떤 단어나 토큰도 아닐 경우를 나타냅니다.

코드 6-11 GloVe 단어 임베딩 행렬 준비하기

```
embedding_dim = 100

embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
    if i < max_words:
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector -------- 임베딩 인덱스에 없는 단어는 모두 0이 됩니다.
```

모델 정의하기

이전과 동일한 구조의 모델을 사용하겠습니다.

코드 6-12 모델 정의하기

```
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()
```

모델에 GloVe 임베딩 로드하기

Embedding 층은 하나의 가중치 행렬을 가집니다. 이 행렬은 2D 부동 소수 행렬이고 각 i번째 원소는 i번째 인덱스에 상응하는 단어 벡터입니다. 간단하네요. 모델의 첫 번째 층인 Embedding 층에 준비된 GloVe 행렬을 로드하세요.[16]

코드 6-13 사전 훈련된 단어 임베딩을 Embedding 층에 로드하기

```
model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False
```

추가적으로 Embedding 층을 동결합니다(trainable 속성을 False로 설정합니다). 사전 훈련된 컨브넷 특성을 사용할 때와 같은 이유입니다. 모델의 일부는 (Embedding 층처럼) 사전 훈련되고 다른 부분은 (최상단 분류기처럼) 랜덤하게 초기화되었다면 훈련하는 동안 사전 훈련된 부분이 업데이트되면 안 됩니다. 이미 알고 있던 정보를 모두 잃게 됩니다. 랜덤하게 초기화된 층에서 대량의 그래디언트 업데이트가 발생하면 이미 학습된 특성을 오염시키기 때문입니다.

모델 훈련과 평가

모델을 컴파일하고 훈련합니다.

코드 6-14 훈련과 평가하기

```
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')
```

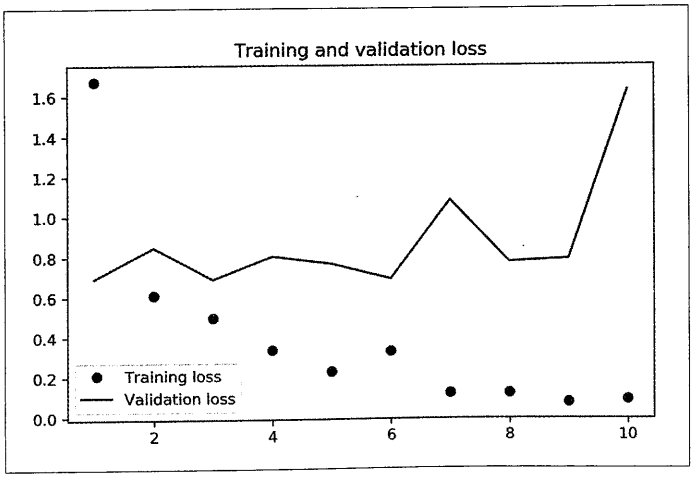
이제 모델의 성능을 그래프로 그려 보겠습니다(그림 6-5와 그림 6-6 참고).

16 옮긴 set_weights() 메서드는 넘파이 배열의 리스트를 매개변수로 받습니다. 전달된 넘파이 배열로 층의 가중치를 설정합니다. 넘파이 배열의 순서와 크기는 층의 weights 속성과 동일해야 합니다. Embedding 층은 가중치가 하나입니다(편향이 없습니다).

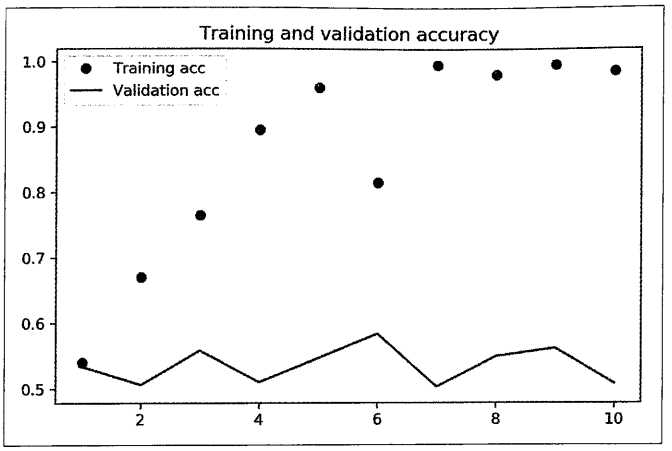
코드 6-15 결과 그래프 그리기

```
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()
```

❮ 그림 6-5 사전 훈련된 단어 임베딩을 사용했을 때 훈련 손실과 검증 손실



❯ 그림 6-6 사전 훈련된 단어 임베딩을 사용했을 때 훈련 정확도와 검증 정확도



이 모델은 과대적합이 빠르게 시작됩니다. 훈련 샘플 수가 작기 때문에 놀라운 일은 아닙니다. 같은 이유로 검증 정확도와 훈련 정확도 사이에 차이가 큽니다. 검증 정확도는 50% 후반을 달성한 것 같습니다.

훈련 샘플 수가 적기 때문에 어떤 샘플 200개를 선택했는지에 따라 성능이 크게 좌우됩니다. 여기서는 샘플들을 랜덤하게 선택했습니다. 선택한 샘플에서 성능이 나쁘면 예제를 위해서 랜덤하게 200개의 샘플을 다시 추출하세요(실전에서는 훈련 데이터를 고르지 않습니다).

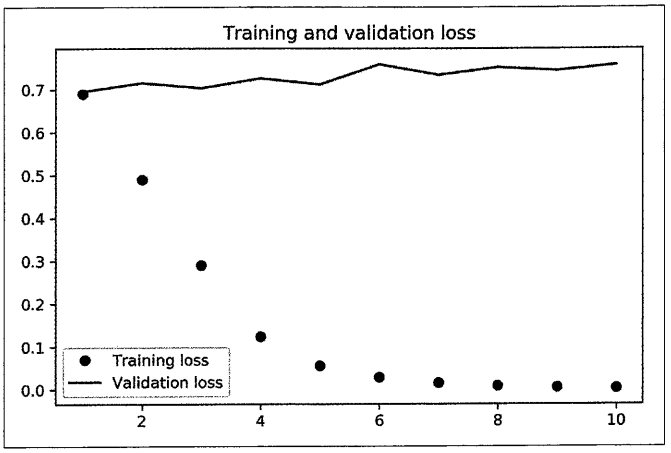
사전 훈련된 단어 임베딩을 사용하지 않거나 임베딩 층을 동결하지 않고 같은 모델을 훈련할 수 있습니다. 이런 경우 해당 작업에 특화된 입력 토큰의 임베딩을 학습할 것입니다. 데이터가 풍부하게 있다면 사전 훈련된 단어 임베딩보다 일반적으로 훨씬 성능이 높습니다. 여기서는 훈련 샘플이 200개뿐이지만 한번 시도해 보죠(그림 6-7과 그림 6-8 참고).

코드 6-16 사전 훈련된 단어 임베딩을 사용하지 않고 같은 모델 훈련하기

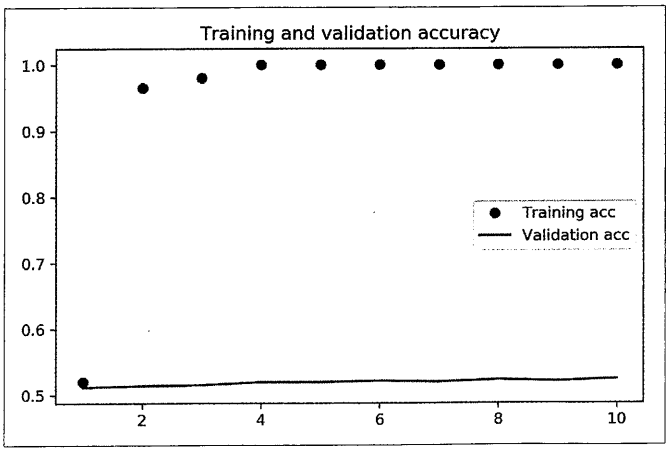
```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

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))
```

❤ 그림 6-7 사전 훈련된 단어 임베딩을 사용하지 않았을 때 훈련 손실과 검증 손실



❤ 그림 6-8 사전 훈련된 단어 임베딩을 사용하지 않았을 때 훈련 정확도와 검증 정확도



검증 정확도는 50% 초반에 멈추어 있습니다. 이 예제에서는 사전 훈련된 단어 임베딩을 사용하는 것이 임베딩을 함께 훈련하는 것보다 낫습니다. 훈련 샘플의 수를 늘리면 금세 상황이 바뀝니다. 연습 삼아 한번 확인해 보세요.[17]

마지막으로 테스트 데이터에서 모델을 평가해 보죠. 먼저 테스트 데이터를 토큰화해야 합니다.

코드 6-17 테스트 데이터 토큰화하기

```
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), encoding="utf8")
            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)
```

그다음 이 절의 첫 번째 모델을 로드하고 평가합니다.[18]

코드 6-18 테스트 세트에서 모델 평가하기

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

테스트 정확도는 겨우 50% 정도입니다. 적은 수의 훈련 샘플로 작업하는 것은 어려운 일이군요!

17 옮긴 샘플 개수를 2,000개 정도 사용하면 사전 훈련된 단어 임베딩을 사용하지 않고도 70%에 가까운 검증 정확도를 얻을 수 있습니다. 테스트 코드와 그래프는 번역서의 깃허브를 참고하세요.

18 옮긴 코드 6-14에서 저장한 모델입니다.

6.1.4 정리

이제 다음 작업을 할 수 있습니다.

- 원본 텍스트를 신경망이 처리할 수 있는 형태로 변환합니다.
- 케라스 모델에 `Embedding` 층을 추가하여 어떤 작업에 특화된 토큰 임베딩을 학습합니다.
- 데이터가 부족한 자연어 처리 문제에서 사전 훈련된 단어 임베딩을 사용하여 성능 향상을 꾀합니다.

DEEP LEARNING

6.2 순환 신경망 이해하기

완전 연결 네트워크나 컨브넷처럼 지금까지 본 모든 신경망의 특징은 메모리가 없다는 것입니다. 네트워크에 주입되는 입력은 개별적으로 처리되며 입력 간에 유지되는 상태가 없습니다. 이런 네트워크로 시퀀스나 시계열 데이터 포인트를 처리하려면 네트워크에 전체 시퀀스를 주입해야 합니다. 즉 전체 시퀀스를 하나의 데이터 포인트로 변환해야 합니다. 예를 들어 IMDB 문제에서 영화 리뷰 하나를 큰 벡터 하나로 변환하여 처리했습니다. 이런 네트워크를 **피드포워드 네트워크**(feedforward network)라고 합니다.[19]

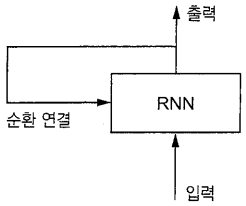
이와 반대로 사람이 문장을 읽는 것처럼 이전에 나온 것을 기억하면서 단어별로 또는 한눈에 들어오는 만큼씩 처리할 수 있습니다. 이는 문장에 있는 의미를 자연스럽게 표현하도록 도와줍니다. 생물학적 지능은 정보 처리를 위한 내부 모델을 유지하면서 점진적으로 정보를 처리합니다. 이 모델은 과거 정보를 사용하여 구축되며 새롭게 얻은 정보를 계속 업데이트합니다.

비록 극단적으로 단순화시킨 버전이지만 **순환 신경망**(Recurrent Neural Network, RNN)은 같은 원리를 적용한 것입니다. 시퀀스의 원소를 순회하면서 지금까지 처리한 정보를 **상태**(state)에 저장합니다. 사실 RNN은 내부에 루프(loop)를 가진 신경망의 한 종류입니다(그림 6-9 참고). RNN의 상태는 2개의 다른 시퀀스(2개의 다른 IMDB 리뷰)를 처리하는 사이에 재설정됩니다. 하나의 시퀀스

19 옮긴 피드포워드 신경망은 완전 연결 신경망과 합성곱 신경망을 포함합니다. 하지만 종종 피드포워드 신경망과 완전 연결 신경망을 동일하게 취급하는 경우가 많습니다.

가 여전히 하나의 데이터 포인트로 간주됩니다. 즉 네트워크에 하나의 입력을 주입한다고 가정합니다. 이 데이터 포인트가 한 번에 처리되지 않는다는 것이 다릅니다. 그 대신 네트워크는 시퀀스의 원소를 차례대로 방문합니다.

❮ 그림 6-9 순환 네트워크: 루프를 가진 네트워크



루프와 상태에 대한 개념을 명확히 하기 위해 넘파이로 간단한 RNN 정방향 계산을 구현해 보죠. 이 RNN은 크기가 (timesteps, input_features)인 2D 텐서로 인코딩된 벡터의 시퀀스를 입력받습니다. 이 시퀀스는 타임스텝을 따라서 반복됩니다. 각 타임스텝 t에서 현재 상태와 ((input_features,) 크기의) 입력을 연결하여 출력을 계산합니다. 그다음 이 출력을 다음 스텝의 상태로 설정합니다. 첫 번째 타임스텝에서는 이전 출력이 정의되지 않으므로 현재 상태가 없습니다. 이때는 네트워크의 **초기 상태**(initial state)인 0 벡터로 상태를 초기화합니다.

의사코드(pseudocode)로 표현하면 RNN은 다음과 같습니다.

코드 6-19 의사코드로 표현한 RNN

```
state_t = 0 -------- 타임스텝 t의 상태입니다.
for input_t in input_sequence: -------- 시퀀스의 원소를 반복합니다.
    output_t = f(input_t, state_t)
    state_t = output_t -------- 출력은 다음 반복을 위한 상태가 됩니다.
```

f 함수는 입력과 상태를 출력으로 변환합니다. 이를 2개의 행렬 W와 U 그리고 편향 벡터를 사용하는 변환으로 바꿀 수 있습니다. 피드포워드 네트워크의 완전 연결 층에서 수행되는 변환과 비슷합니다.

코드 6-20 좀 더 자세한 의사코드로 표현한 RNN

```
state_t = 0
for input_t in input_sequence:
    output_t = activation(dot(W, input_t) + dot(U, state_t) + b)
    state_t = output_t
```

완벽하게 설명하기 위해 간단한 RNN의 정방향 계산을 넘파이로 구현해 보죠.

코드 6-21 넘파이로 구현한 간단한 RNN

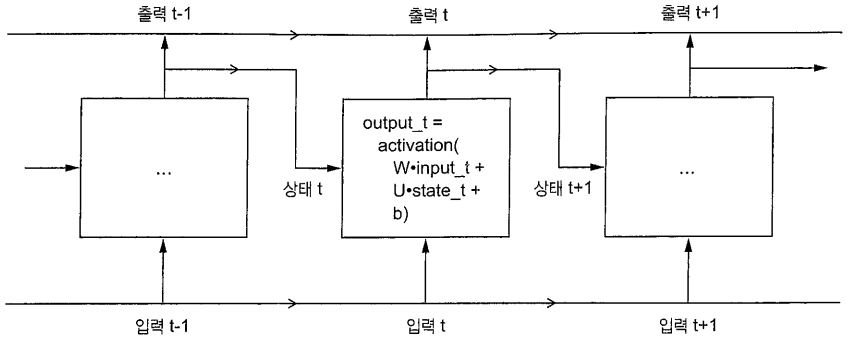
```
import numpy as np

timesteps = 100 -------- 입력 시퀀스에 있는 타임스텝의 수
input_features = 32 -------- 입력 특성의 차원
output_features = 64 -------- 출력 특성의 차원

inputs = np.random.random((timesteps, input_features)) -------- 입력 데이터: 예제를 위해 생성한 난수

state_t = np.zeros((output_features,)) -------- 초기 상태: 모두 0인 벡터

W = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features)) ---- 랜덤한 가중치 행렬을 만듭니다.
b = np.random.random((output_features,))

successive_outputs = []
for input_t in inputs: -------- input_t는 크기가 (input_features,)인 벡터입니다.
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
                                          -------- 입력과 현재 상태(이전 출력)를 연결하여 현재 출력을 얻습니다.
    successive_outputs.append(output_t) -------- 이 출력을 리스트에 저장합니다.

    state_t = output_t -------- 다음 타임스텝을 위해 네트워크의 상태를 업데이트합니다.[20]

                                -------- 최종 출력은 크기가 (timesteps, output_features)인 2D 텐서입니다.
final_output_sequence = np.stack(successive_outputs, axis=0)
```

아주 쉽네요. 요약하면 RNN은 반복할 때 이전에 계산한 정보를 재사용하는 for 루프에 지나지 않습니다. 물론 이 정의에 맞는 RNN의 종류는 많습니다. 이 예는 가장 간단한 RNN의 형태입니다. RNN은 스텝(step) 함수에 의해 특화됩니다. 이 예에서는 다음과 같습니다(그림 6-10 참고).

```
output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
```

20 옮긴 기본 RNN의 상태를 은닉 상태(hidden state)라고도 부릅니다. 은닉 상태는 이전 타임스텝의 출력입니다.

❤ 그림 6-10 시간에 따라 펼쳐진 간단한 RNN[21]



Note ≡ 이 예에서 최종 출력은 (timesteps, output_features) 크기의 2D 텐서입니다. 각 타임스텝은 시간 t에서의 출력을 나타냅니다. 출력 텐서의 각 타임스텝 t에는 입력 시퀀스에 있는 타임스텝 0에서 t까지 전체 과거에 대한 정보를 담고 있습니다. 이런 이유 때문에 많은 경우 전체 출력 시퀀스가 필요하지 않습니다. 전체 시퀀스에 대한 정보를 이미 담고 있으므로 마지막 출력(루프의 마지막 output_t)만 있으면 됩니다.

6.2.1 케라스의 순환 층

넘파이로 간단하게 구현한 과정이 실제 케라스의 SimpleRNN 층에 해당합니다.

```
from keras.layers import SimpleRNN
```

SimpleRNN이 한 가지 다른 점은 넘파이 예제처럼 하나의 시퀀스가 아니라 다른 케라스 층과 마찬가지로 시퀀스 배치를 처리한다는 것입니다. 즉 (timesteps, input_features) 크기가 아니라 (batch_size, timesteps, input_features) 크기의 입력을 받습니다.

케라스에 있는 모든 순환 층과 마찬가지로 SimpleRNN은 두 가지 모드로 실행할 수 있습니다. 각 타임스텝의 출력을 모은 전체 시퀀스를 반환하거나(크기가 (batch_size, timesteps, output_features)인 3D 텐서), 입력 시퀀스에 대한 마지막 출력만 반환할 수 있습니다(크기가 (batch_size, output_features)인 2D 텐서). 이 모드는 객체를 생성할 때 return_sequences 매개변

21 역주 이 그림은 그림 6-9와 같은 순환 네트워크를 타임스텝에 따라 실행되는 모습을 표현한 것입니다. RNN은 종종 이와 같이 시간에 따라 펼친 그림으로 자주 나타냅니다. 이런 표현 때문에 타임스텝마다 다른 W, U, b가 있다고 오해하기 쉽습니다. 코드 6-21에서 볼 수 있듯이 RNN은 모든 타임스텝에 걸쳐 동일한 W, U, b가 사용됩니다.

수로 선택할 수 있습니다. SimpleRNN을 사용하여 마지막 타임스텝의 출력만 얻는 예제를 살펴보죠.[22]

```
>>> from keras.models import Sequential
>>> from keras.layers import Embedding, SimpleRNN
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32))
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, None, 32)          320000
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, 32)                2080
=================================================================
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
```

다음 예는 전체 상태 시퀀스를 반환합니다.[23]

```
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, None, 32)          320000
_________________________________________________________________
simple_rnn_2 (SimpleRNN)     (None, None, 32)          2080
=================================================================
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
```

22 역주 SimpleRNN의 입력 특성 크기는 Embedding 층의 출력 특성 크기가 되므로 입력 특성과 출력 특성의 크기가 모두 32입니다. 전체 가중치 파라미터의 수는 (32, 32) 크기의 W, U와 (32,) 크기의 b를 더하여 2,080이 됩니다.

23 역주 층의 이름은 클래스 이름에서 단어 사이에 밑줄 문자를 추가하고 소문자로 바꾸어 사용합니다(snake case). 이름 뒤에 붙는 숫자는 클래스별로 1씩 증가됩니다. 별도의 이름을 주려면 SimpleRNN(32, name='my_first_rnn')처럼 name 매개변수를 사용합니다.

네트워크의 표현력을 증가시키기 위해 여러 개의 순환 층을 차례대로 쌓는 것이 유용할 때가 있습니다. 이런 설정에서는 중간층들이 전체 출력 시퀀스를 반환하도록 설정해야 합니다.

```
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32)) -------- 맨 위 층만 마지막 출력을 반환합니다.
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, None, 32)          320000
_________________________________________________________________
simple_rnn_3 (SimpleRNN)     (None, None, 32)          2080
_________________________________________________________________
simple_rnn_4 (SimpleRNN)     (None, None, 32)          2080
_________________________________________________________________
simple_rnn_5 (SimpleRNN)     (None, None, 32)          2080
_________________________________________________________________
simple_rnn_6 (SimpleRNN)     (None, 32)                2080
=================================================================
Total params: 328,320
Trainable params: 328,320
Non-trainable params: 0
```

이제 IMDB 영화 리뷰 분류 문제에 적용해 보죠. 먼저 데이터를 전처리합니다.

코드 6-22 IMDB 데이터 전처리하기

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

max_features = 10000 -------- 특성으로 사용할 단어의 수
maxlen = 500 -------- 사용할 텍스트의 길이(가장 빈번한 max_features개의 단어만 사용합니다.)
batch_size = 32

print('데이터 로딩 ... ')
(input_train, y_train), (input_test, y_test) = imdb.load_data(
    num_words=max_features)
print(len(input_train), '훈련 시퀀스')
print(len(input_test), '테스트 시퀀스')

print('시퀀스 패딩 (samples x time)')
```

```
input_train = sequence.pad_sequences(input_train, maxlen=maxlen)
input_test = sequence.pad_sequences(input_test, maxlen=maxlen)
print('input_train 크기:', input_train.shape)
print('input_test 크기:', input_test.shape)
```

Embedding 층과 SimpleRNN 층을 사용하여 간단한 순환 네트워크를 훈련시켜 보겠습니다.

코드 6-23 Embedding 층과 SimpleRNN 층을 사용한 모델 훈련하기

```
from keras.layers import Dense

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

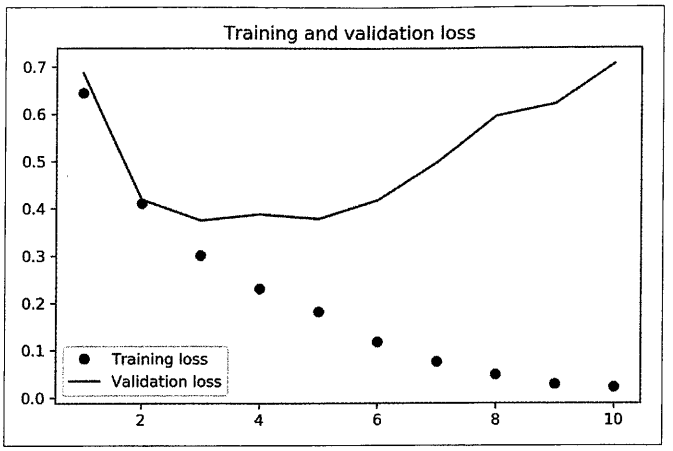
이제 훈련과 검증의 손실과 정확도를 그래프로 그립니다(그림 6-11과 그림 6-12 참고).

코드 6-24 결과 그래프 그리기

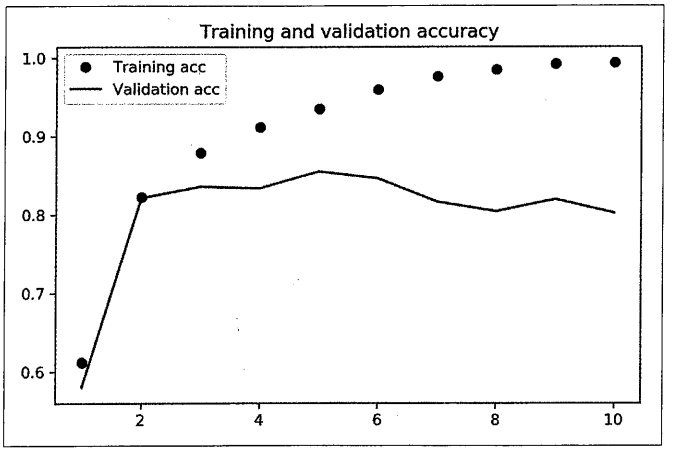
```
import matplotlib.pyplot as plt

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()
```

❯ 그림 6-11 SimpleRNN을 사용한 IMDB 문제의 훈련 손실과 검증 손실



❯ 그림 6-12 SimpleRNN을 사용한 IMDB 문제의 훈련 정확도와 검증 정확도



3장에서 이 데이터셋을 사용한 첫 번째 모델에서 얻은 테스트 정확도는 87%였습니다. 안타깝지만 간단한 순환 네트워크는 이 기준 모델보다 성능이 높지 않습니다(85% 정도의 검증 정확도를 얻었습니다). 이런 원인은 전체 시퀀스가 아니라 순서대로 500개의 단어만 입력에 사용했기 때문입니다. 이 RNN은 기준 모델보다 얻은 정보가 적습니다. 다른 이유는 SimpleRNN이 텍스트처럼

긴 시퀀스를 처리하는 데 적합하지 않기 때문입니다. 더 잘 작동하는 다른 순환 층이 있습니다. 조금 더 고급 순환 층을 살펴보죠.

6.2.2 LSTM과 GRU 층 이해하기

케라스에는 SimpleRNN 외에 다른 순환 층도 있습니다. LSTM과 GRU 2개입니다. 실전에서는 항상 이 둘 중에 하나를 사용할 것입니다. SimpleRNN은 실전에 쓰기에는 너무 단순하기 때문입니다. SimpleRNN은 이론적으로 시간 t에서 이전의 모든 타임스텝의 정보를 유지할 수 있습니다. 실제로는 긴 시간에 걸친 의존성은 학습할 수 없는 것이 문제입니다. 층이 많은 일반 네트워크(피드포워드 네트워크)에서 나타나는 것과 비슷한 현상인 **그래디언트 소실 문제**(vanishing gradient problem) 때문입니다.[24] 피드포워드 네트워크에 층을 많이 추가할수록 훈련하기 어려운 것과 같습니다. 1990년대 초 호크라이터(Hochreiter), 슈미트후버(Schmidhuber), 벤지오(Bengio)가 이런 현상에 대한 이론적인 원인을 연구했습니다.[25] 이 문제를 해결하기 위해 고안된 것이 LSTM과 GRU 층입니다.

LSTM 층을 살펴보죠. 장 · 단기 메모리(Long Short-Term Memory, LSTM) 알고리즘은 호크라이터와 슈미트후버가 1997년에 개발했습니다.[26] 이 알고리즘은 그래디언트 소실 문제에 대한 연구의 결정체입니다.

이 층은 앞서 보았던 SimpleRNN의 한 변종입니다. 정보를 여러 타임스텝에 걸쳐 나르는 방법이 추가됩니다. 처리할 시퀀스에 나란히 작동하는 컨베이어 벨트를 생각해 보세요. 시퀀스 어느 지점에서 추출된 정보가 컨베이어 벨트 위로 올라가 필요한 시점의 타임스텝으로 이동하여 떨굽니다. 이것이 LSTM이 하는 일입니다. 나중을 위해 정보를 저장함으로써 처리 과정에서 오래된 시그널의 점차 소실되는 것을 막아 줍니다.

이를 자세하게 이해하기 위해 SimpleRNN 셀(cell)[27]부터 그려 보겠습니다(그림 6-13 참고). 가중치 행렬 여러 개가 나오므로 출력(output)을 나타내는 문자 o로 셀에 있는 W와 U 행렬을 표현하겠습니다(Wo와 Uo).

---

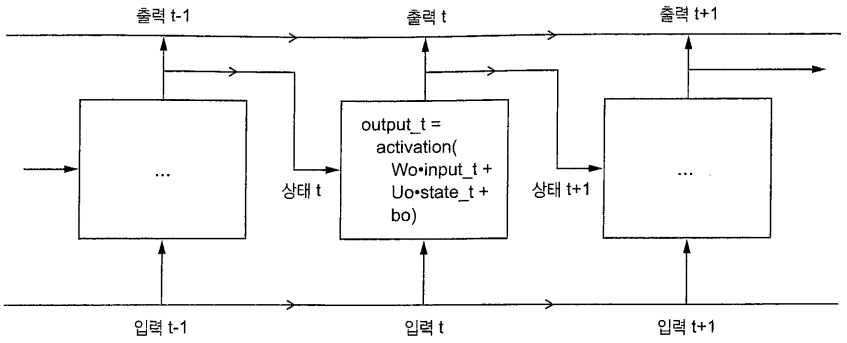
24 역주 순환 신경망의 역전파는 그림 6-13과 같이 타임스텝에 따라 네트워크가 펼쳐진 것처럼 진행됩니다. 이를 BPTT(BackPropagation Through Time)라고 합니다. 역전파가 진행되는 동안 타임스텝마다 동일한 가중치를 사용하기 때문에 타임스텝이 길어질수록 그래디언트 값이 급격히 줄어들거나(vanishing gradient problem) 급격히 증가할 수 있습니다(exploding gradient problem).

25 예를 들어 다음을 참고하세요. Yoshua Bengio, Patrice Simard, and Paolo Frasconi, "Learning Long-Term Dependencies with Gradient Descent Is Difficult," IEEE Transactions on Neural Networks 5, no. 2 (1994).

26 Sepp Hochreiter and Jürgen Schmidhuber, "Long Short-Term Memory," Neural Computation 9, no. 8 (1997).

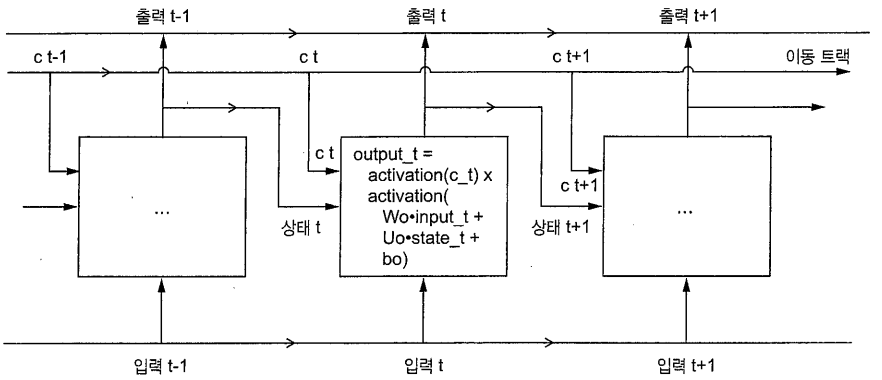
27 역주 RNN에서는 층 또는 뉴런을 종종 셀(cell)이라고 표현합니다.

❤ 그림 6-13 LSTM 층의 시작점: SimpleRNN



이 그림에 타임스텝을 가로질러 정보를 나르는 데이터 흐름을 추가해 보죠. 타임스텝 t에서 이 값을 이동 상태 c_t라고 부르겠습니다. 여기서 c는 이동(carry)을 의미합니다.[28] 이 정보를 사용하여 셀이 다음과 같이 바뀝니다. 입력 연결과 순환 연결(상태)로부터 이 정보가 합성됩니다(완전 연결 층과 같은 변환: 가중치 행렬과 점곱한 후 편향을 더하고 활성화 함수를 적용합니다). 그러고는 다음 타임스텝으로 전달될 상태를 변경시킵니다(활성화 함수와 곱셈을 통해서). 개념적으로 보면 데이터를 실어 나르는 이 흐름이 다음 출력과 상태를 조절합니다(그림 6-14 참고). 여기까지는 간단합니다.

❤ 그림 6-14 SimpleRNN에서 LSTM으로: 이동 트랙 추가



28 영문 일반적으로 이를 셀 상태(cell state)라고 부릅니다. 번역서에서는 혼동을 줄이기 위해 RNN의 은닉 상태를 그냥 상태로, 셀 상태를 이동 상태로 부르겠습니다.

이제 복잡한 부분은 데이터 흐름에서 다음 이동 상태(c_t+1)가 계산되는 방식입니다. 여기에는 3개의 다른 변환이 관련되어 있습니다. 3개 모두 SimpleRNN과 같은 형태를 가집니다.

```
y = activation(dot(state_t, U) + dot(input_t, W) + b)
```

3개의 변환 모두 자신만의 가중치 행렬을 가집니다. 각각 i, f, k로 표시하겠습니다. 다음이 지금까지 설명한 내용입니다(약간 이상하게 보일 수 있지만 곧 설명하니 조금만 기다려 주세요).

코드 6-25 LSTM 구조의 의사코드(1/2)[29]

```
output_t = activation(c_t) * activation(dot(input_t, Wo) + dot(state_t, Uo) + bo)

i_t = activation(dot(state_t, Ui) + dot(input_t, Wi) + bi)
f_t = activation(dot(state_t, Uf) + dot(input_t, Wf) + bf)
k_t = activation(dot(state_t, Uk) + dot(input_t, Wk) + bk)
```

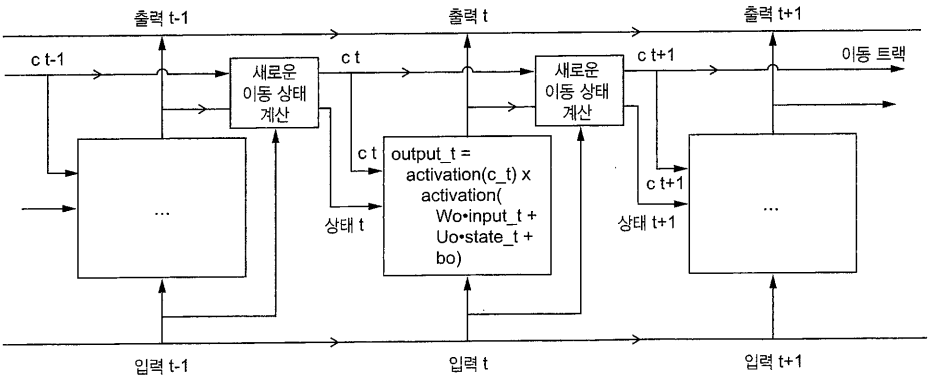
i_t, f_t, k_t를 결합하여 새로운 이동 상태(c_t+1)를 구합니다.

코드 6-26 LSTM 구조의 의사코드(2/2)

```
c_t + 1 = i_t * k_t + c_t * f_t
```

그림 6-15에 이를 추가했습니다. 이것이 전부입니다. 아주 복잡하지 않네요. 조금 까다로울 뿐입니다.

❤ **그림 6-15** LSTM 구조



29 **역주** output_t에 적용되는 활성화 함수는 순서대로 tanh와 sigmoid 함수입니다. i_t, f_t에는 sigmoid 활성화 함수가 적용되고 k_t에는 tanh 활성화 함수가 적용됩니다.

이 연산들이 하는 일을 해석하면 각 의미에 대해 통찰을 얻을 수 있습니다. 예를 들어 c_t와 f_t의 곱셈은 이동을 위한 데이터 흐름에서 관련이 적은 정보를 의도적으로 삭제한다고 볼 수 있습니다. 한편 i_t와 k_t는 현재에 대한 정보를 제공하고 이동 트랙을 새로운 정보로 업데이트합니다.[30] 하지만 결국 이런 해석은 큰 의미가 없습니다. 이 연산들이 실제로 하는 일은 연산에 관련된 가중치 행렬에 따라 결정되기 때문입니다. 이 가중치는 엔드-투-엔드 방식으로 학습됩니다. 이 과정은 훈련 반복마다 매번 새로 시작되며 이런저런 연산들에 특정 목적을 부여하기가 불가능합니다. RNN 셀의 사양(specification)은 가설 공간을 결정합니다. 훈련할 때 이 공간에서 좋은 모델 파라미터를 찾습니다. 셀의 사양이 셀이 하는 일을 결정하지 않습니다. 이는 셀의 가중치에 달려 있습니다. 같은 셀이더라도 다른 가중치를 가지는 경우 매우 다른 작업을 수행합니다. 따라서 RNN 셀을 구성하는 연산 조합은 엔지니어링적인 설계가 아니라 가설 공간의 제약 조건으로 해석하는 것이 낫습니다.

연구자에게는 RNN 셀의 구현 방법 같은 제약 조건의 선택을 엔지니어보다 (유전 알고리즘이나 강화 학습 알고리즘 같은) 최적화 알고리즘에 맡기면 더 나아 보일 것입니다. 미래에는 이런 식으로 네트워크를 만들게 될 것입니다. 요약하면 LSTM 셀의 구체적인 구조에 대해 이해할 필요가 전혀 없습니다. 우리가 해야 할 일도 아닙니다. LSTM 셀의 역할만 기억하면 됩니다. 바로 과거 정보를 나중에 다시 주입하여 그래디언트 소실 문제를 해결하는 것입니다.

6.2.3 케라스를 사용한 LSTM 예제

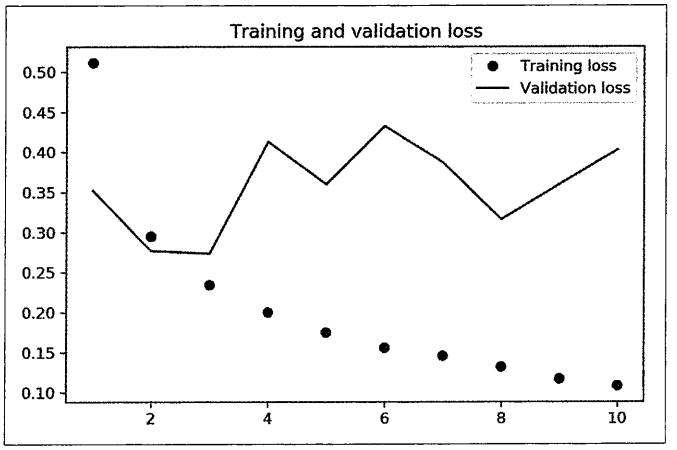
이제 실제적인 관심사로 이동해 보죠. LSTM 층으로 모델을 구성하고 IMDB 데이터에서 훈련해 보겠습니다(그림 6-16과 그림 6-17 참고). 이 네트워크는 조금 전 SimpleRNN을 사용했던 모델과 비슷합니다. LSTM 층은 출력 차원만 지정하고 다른 (많은) 매개변수는 케라스의 기본값으로 남겨 두었습니다. 케라스는 좋은 기본값을 가지고 있어서 직접 매개변수를 튜닝하는 데 시간을 쓰지 않고도 거의 항상 어느 정도 작동하는 모델을 얻을 수 있습니다.

30 역주 이런 해석 때문에 f_t의 계산식을 삭제 게이트(forget gate), i_t의 계산식을 입력 게이트(input gate)라고 부릅니다. 또 output_t의 계산식을 출력 게이트(output gate)라고 부릅니다.

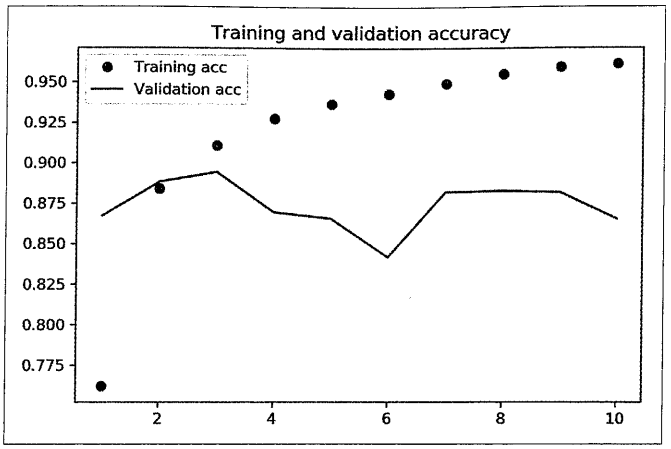
코드 6-27 케라스에서 LSTM 층 사용하기

```
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

❮ 그림 6-16 LSTM을 사용한 IMDB 문제의 훈련 손실과 검증 손실



❤ 그림 6-17 LSTM을 사용한 IMDB 문제의 훈련 정확도와 검증 정확도



이번에는 88% 정도의 검증 정확도를 달성했습니다. 나쁘지 않네요. SimpleRNN 네트워크보다 확실히 더 낫습니다. LSTM이 그래디언트 소실 문제로부터 덜 영향을 받기 때문입니다. 3장에서 사용했던 것보다 적은 데이터를 사용하고도 3장의 완전 연결 네트워크보다 조금 더 낮습니다. 3장에서는 전체 시퀀스를 사용했지만 여기서는 500 타임스텝 이후의 시퀀스를 잘라 버렸습니다.

하지만 많은 계산을 사용한 것치고 획기적인 결과는 아닙니다. 왜 LSTM의 성능이 더 높지 않을까요? 한 가지 이유는 임베딩 차원이나 LSTM 출력 차원 같은 하이퍼파라미터를 전혀 튜닝하지 않았기 때문입니다. 또 하나는 규제가 없기 때문입니다. 솔직히 말하면 가장 큰 이유는 리뷰를 전체적으로 길게 분석하는 것(LSTM이 잘하는 일)은 감성 분류 문제에 도움이 되지 않기 때문입니다. 이런 간단한 문제는 각 리뷰에 어떤 단어가 나타나고 얼마나 등장하는지를 보는 것이 낫습니다. 바로 첫 번째 완전 연결 네트워크가 사용한 방법입니다. 하지만 훨씬 더 복잡한 자연어 처리 문제들에서는 LSTM 능력이 드러납니다. 특히 질문-응답(question-answering)과 기계 번역(machine translation) 분야입니다.

6.2.4 정리

이제 다음을 알게 되었습니다.

- RNN이 무엇이고 동작하는 방법
- LSTM이 무엇이고 긴 시퀀스에서 단순한 RNN보다 더 잘 작동하는 이유
- 케라스의 RNN 층을 사용하여 시퀀스 데이터를 처리하는 방법

다음으로 RNN의 여러 고급 기능을 살펴보겠습니다. 이런 기능을 사용하면 딥러닝의 시퀀스 모델을 최대로 활용할 수 있습니다.

DEEP LEARNING

6.3 순환 신경망의 고급 사용법

이 절에서는 순환 신경망의 성능과 일반화 능력을 향상시키기 위한 세 가지 고급 기술을 살펴보겠습니다. 이 절을 마칠 때면 케라스에서 순환 신경망을 사용하는 대부분의 방법을 알게 될 것입니다. 온도 예측 문제로 세 가지 개념을 모두 시연해 보겠습니다. 이 시계열 데이터는 건물 옥상에 설치된 센서에서 취득한 온도, 기압, 습도 같은 데이터입니다. 이 데이터를 사용하여 마지막 데이터 포인트에서 24시간 이후의 온도를 예측하겠습니다. 이 문제는 시계열 데이터에서 일반적으로 나타나는 여러 가지 어려운 점을 가지고 있습니다. 전형적이고 꽤 도전적인 문제입니다.

다음 기법들을 적용하겠습니다.

- **순환 드롭아웃**(recurrent dropout): 순환 층에서 과대적합을 방지하기 위해 케라스에 내장되어 있는 드롭아웃을 사용합니다.
- **스태킹 순환 층**(stacking recurrent layer): 네트워크의 표현 능력(representational power)을 증가시킵니다(그 대신 계산 비용이 많이 듭니다).
- **양방향 순환 층**(bidirectional recurrent layer): 순환 네트워크에 같은 정보를 다른 방향으로 주입하여 정확도를 높이고 기억을 좀 더 오래 유지시킵니다.

6.3.1 기온 예측 문제

지금까지 다룬 시퀀스 데이터는 IMDB 데이터셋이나 로이터 데이터셋처럼 텍스트 데이터입니다. 시퀀스 데이터는 이런 언어 처리 분야뿐만 아니라 훨씬 많은 문제에서 등장합니다. 이 절에 있는 모든 예제는 날씨 시계열 데이터셋을 사용합니다. 이 데이터는 독일 예나(Jena) 시에 있는 막스 플랑크 생물지구화학 연구소(Max Planck Institute for Biogeochemistry)의 지상 관측소에서 수집한 것입니다.[31]

이 데이터셋에는 수년간에 걸쳐 (기온, 기압, 습도, 풍향 등) 14개의 관측치가 10분마다 기록되어 있습니다. 원본 데이터는 2003년부터 기록되어 있지만 이 예제에서는 2009~2016년 사이의 데이터만 사용합니다. 이 데이터셋은 시계열 수치 데이터를 다루는 법을 익히는 데 안성맞춤입니다. 최근 데이터(몇 일치 데이터 포인트)를 입력으로 사용하여 모델을 만들고 24시간 이후의 기온을 예측하겠습니다.

다음과 같이 데이터를 내려받고 압축을 풉니다.[32]

```
> cd ~/Downloads
> mkdir jena_climate
> cd jena_climate
> wget https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
> unzip jena_climate_2009_2016.csv.zip
```

데이터를 살펴보죠.

코드 6-28 예나의 날씨 데이터셋 조사하기

```
import os

data_dir = './datasets/jena_climate'
fname = os.path.join(data_dir, 'jena_climate_2009_2016.csv')

f = open(fname)
data = f.read()
f.close()
```

31 Olaf Kolle, https://www.bgc-jena.mpg.de/wetter.

32 **역주** 번역서의 깃허브는 datasets/jena_climate 폴더에 압축 해제한 날씨 데이터셋을 포함하고 있기 때문에 별도로 내려받지 않아도 됩니다.

```
lines = data.split('\n')
header = lines[0].split(',')
lines = lines[1:]

print(header)
print(len(lines))
```

출력된 줄 수는 42만 551입니다(줄마다 하나의 타임스텝이고 날짜와 14개의 날씨 정보가 레코드입니다). 헤더는 다음과 같습니다.

```
["Date Time",
 "p (mbar)",
 "T (degC)",
 "Tpot (K)",
 "Tdew (degC)",
 "rh (%)",
 "VPmax (mbar)",
 "VPact (mbar)",
 "VPdef (mbar)",
 "sh (g/kg)",
 "H2OC (mmol/mol)",
 "rho (g/m**3)",
 "wv (m/s)",
 "max. wv (m/s)",
 "wd (deg)"]
```

42만 551개의 데이터 전체를 넘파이 배열로 바꿉니다.

코드 6-29 데이터 파싱하기

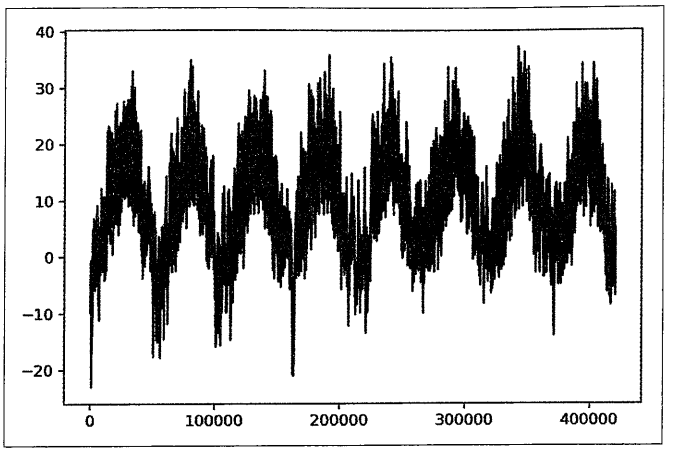
```
import numpy as np

float_data = np.zeros((len(lines), len(header) - 1))
for i, line in enumerate(lines):
    values = [float(x) for x in line.split(',')[1:]]
    float_data[i, :] = values
```

예를 들어 다음은 시간에 따른 기온(섭씨) 그래프입니다(그림 6-18 참고). 이 그래프에서 온도에 주기성이 있다는 것을 잘 볼 수 있습니다.

코드 6-30 시계열 온도 그래프 그리기

```
from matplotlib import pyplot as plt

temp = float_data[:, 1] -------- 온도(섭씨)
plt.plot(range(len(temp)), temp)
```

❤ 그림 6-18 데이터셋 전체 기간의 온도(℃)

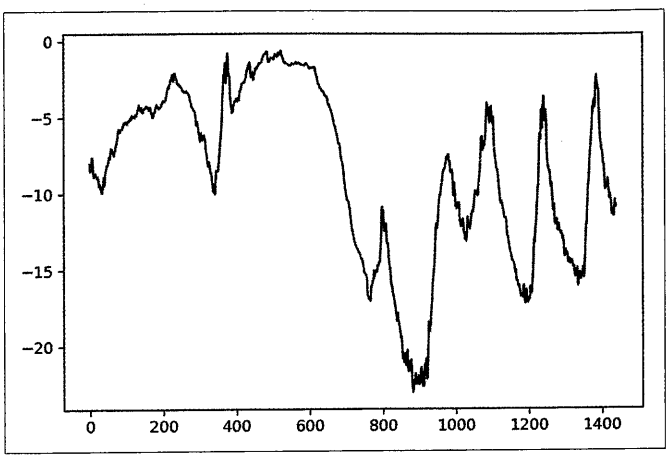


다음은 기간을 좁혀서 처음 10일간 온도 데이터를 나타낸 그래프입니다(그림 6-19 참고). 10분마다 데이터가 기록되므로 하루에 총 144개의 데이터 포인트가 있습니다.

코드 6-31 처음 10일간 온도 그래프 그리기

```
plt.plot(range(1440), temp[:1440])
```

❤ 그림 6-19 데이터셋의 처음 10일간 온도(℃)



이 그래프에서 일별 주기성을 볼 수 있습니다. 특히 마지막 4일간을 보면 확실합니다. 이 데이터는 아주 추운 겨울 중 10일입니다.[33]

지난 몇 달간 데이터를 사용하여 다음 달의 평균 온도를 예측하는 문제는 쉬운 편입니다. 연간 데이터 주기성은 안정적이기 때문입니다. 하지만 하루하루 데이터를 살펴보면 온도 변화는 매우 불안정합니다. 일자별 수준의 시계열 데이터를 예측할 수 있을까요? 직접 확인해 보겠습니다.

6.3.2 데이터 준비

이 문제의 정확한 정의는 다음과 같습니다. lookback 타임스텝(하나의 타임스텝은 10분입니다)만큼 이전으로 돌아가서 매 steps 타임스텝마다 샘플링합니다. 이 데이터를 바탕으로 delay 타임스텝 이후의 온도를 예측할 수 있을까요? 사용할 변수는 다음과 같습니다.

- `lookback = 1440`: 10일 전 데이터로 돌아갑니다.
- `steps = 6`: 1시간마다 데이터 포인트 하나를 샘플링합니다.
- `delay = 144`: 24시간이 지난 데이터가 타깃이 됩니다.

시작하기 전에 두 가지 작업을 처리해야 합니다.

33 옮긴 2009년 1월 1일 0시 10분부터 1월 11일 0시 0분까지 데이터입니다.

- 신경망에 주입할 수 있는 형태로 데이터를 전처리합니다. 데이터가 이미 수치형이므로 추가적인 벡터화가 필요하지 않습니다. 하지만 데이터에 있는 각 시계열 특성의 범위가 서로 다릅니다(예를 들어 온도는 일반적으로 -20도에서 +30도 사이고, 밀리바(mb)로 측정된 기압은 1,000 근처의 값입니다). 각 시계열 특성을 개별적으로 정규화하여 비슷한 범위를 가진 작은 값으로 바꾸겠습니다.
- float_data 배열을 받아 과거 데이터의 배치와 미래 타깃 온도를 추출하는 파이썬 제너레이터(generator)를 만듭니다. 이 데이터셋에 있는 샘플은 중복이 많습니다(샘플 N과 샘플 N + 1은 대부분 타임스텝이 비슷합니다).[34] 모든 샘플을 각기 메모리에 적재하는 것은 낭비가 심하므로 대신에 원본 데이터를 사용하여 그때그때 배치를 만들겠습니다.

각 시계열 특성에 대해 평균을 빼고 표준 편차로 나누어 전처리합니다. 처음 20만 개 타임스텝을 훈련 데이터로 사용할 것이므로 전체 데이터에서 20만 개만 사용하여 평균과 표준 편차를 계산합니다.[35]

코드 6-32 데이터 정규화하기

```
mean = float_data[:200000].mean(axis=0)
float_data -= mean
std = float_data[:200000].std(axis=0)
float_data /= std
```

코드 6-33은 여기서 사용할 제너레이터입니다. 이 제너레이터 함수는 (samples, targets) 튜플을 반복적으로 반환합니다.[36] samples는 입력 데이터로 사용할 배치고 targets는 이에 대응되는 타깃 온도의 배열입니다. 이 제너레이터 함수에는 다음 매개변수가 있습니다.

- data: 코드 6-32에서 정규화한 부동 소수 데이터로 이루어진 원본 배열
- lookback: 입력으로 사용하기 위해 거슬러 올라갈 타임스텝
- delay: 타깃으로 사용할 미래의 타임스텝
- min_index와 max_index: 추출할 타임스텝의 범위를 지정하기 위한 data 배열의 인덱스. 검증 데이터와 테스트 데이터를 분리하는 데 사용합니다.
- shuffle: 샘플을 섞을지, 시간 순서대로 추출할지를 결정합니다.

34 역주 훈련 샘플 하나는 훈련 데이터 범위 안에서 랜덤하게 선택된 후 과거 10일치 데이터 포인트 240개를 사용합니다. 훈련하는 동안 많은 샘플을 추출하기 때문에 모두 넘파이 배열로 만들면 메모리를 많이 차지하고 중복된 포인트도 많습니다.

35 역주 훈련 데이터에서 얻은 통계 값으로 검증 데이터와 테스트 데이터를 전처리해야 합니다.

36 역주 samples의 크기는 (128, 240, 14)이고 targets의 크기는 (128,)입니다. 첫 번째 차원은 배치 차원입니다.

- batch_size: 배치의 샘플 수
- step: 데이터를 샘플링할 타임스텝 간격. 1시간에 하나의 데이터 포인트를 추출하기 위해 6으로 지정하겠습니다.

코드 6-33 시계열 데이터와 타깃을 반환하는 제너레이터 함수[37]

```
def generator(data, lookback, delay, min_index, max_index,
              shuffle=False, batch_size=128, step=6):
    if max_index is None:
        max_index = len(data) - delay - 1
    i = min_index + lookback
    while 1:
        if shuffle:
            rows = np.random.randint(
                min_index + lookback, max_index, size=batch_size)
        else:
            if i + batch_size >= max_index:
                i = min_index + lookback
            rows = np.arange(i, min(i + batch_size, max_index))
            i += len(rows)

        samples = np.zeros((len(rows),
                            lookback // step,
                            data.shape[-1]))
        targets = np.zeros((len(rows),))
        for j, row in enumerate(rows):
            indices = range(rows[j] - lookback, rows[j], step)
            samples[j] = data[indices]
            targets[j] = data[rows[j] + delay][1]
        yield samples, targets
```

이제 generator 함수를 사용하여 훈련용, 검증용, 테스트용으로 3개의 제너레이터를 만들어 보죠. 각 제너레이터는 원본 데이터에서 다른 시간대를 사용합니다. 훈련 제너레이터는 처음 20만 개 타임스텝을 사용하고, 검증 제너레이터는 그다음 10만 개를 사용하고, 테스트 제너레이터는 나머지를 사용합니다.

37 옮긴 이 함수는 훈련할 때 랜덤하게 샘플 시퀀스를 추출하기 위해 shuffle 매개변수를 제공합니다. 전체 훈련 데이터를 순서대로 학습하는 것보다 랜덤하게 선택한 샘플을 사용하여 가설 공간의 다양성을 탐색하면 지역 최솟값을 벗어나기 쉽고 데이터셋에 과대적합되는 것을 줄일 수 있습니다. 이것이 확률적 경사 하강법(SGD)의 핵심 아이디어입니다. 보통은 에포크를 시작하기 전에 전체 데이터셋을 무작위로 섞지만 예나 온도 데이터셋은 시계열 데이터이기 때문에 전체를 섞는 대신 임의의 위치에서 시퀀스를 추출했습니다.

코드 6-34 훈련, 검증, 테스트 제너레이터 준비하기

```
lookback = 1440
step = 6
delay = 144
batch_size = 128
train_gen = generator(float_data,
                      lookback=lookback,
                      delay=delay,
                      min_index=0,
                      max_index=200000,
                      shuffle=True,
                      step=step,
                      batch_size=batch_size)
val_gen = generator(float_data,
                    lookback=lookback,
                    delay=delay,
                    min_index=200001,
                    max_index=300000,
                    step=step,
                    batch_size=batch_size)
test_gen = generator(float_data,
                     lookback=lookback,
                     delay=delay,
                     min_index=300001,
                     max_index=None,
                     step=step,
                     batch_size=batch_size)
                                          ┌------ 전체 검증 세트를 순회하기 위해 val_gen에서 추출할 횟수[38]
val_steps = (300000 - 200001 - lookback) // batch_size
                                          ┌------ 전체 테스트 세트를 순회하기 위해 test_gen에서 추출할 횟수
test_steps = (len(float_data) - 300001 - lookback) // batch_size
```


6.3.3 상식 수준의 기준점

블랙 박스 같은 딥러닝 모델을 사용하여 온도 예측 문제를 풀기 전에 간단한 상식 수준의 해법을 시도해 보겠습니다. 이는 정상 여부 확인을 위한 용도고 고수준 머신 러닝 모델이라면 뛰어넘어

38 옮긴 generator() 함수는 while 문을 사용하여 무한 반복되기 때문에 검증 세트와 테스트 세트를 한 번 순회하는 횟수를 알려 주어야 합니다.

야 할 기준점을 만듭니다. 이런 상식 수준의 해법은 알려진 해결책이 없는 새로운 문제를 다루어야 할 때 유용합니다. 일부 클래스가 월등히 많아 불균형한 분류 문제가 고전적인 예입니다. 데이터셋에 클래스 A의 샘플이 90%, 클래스 B의 샘플이 10%가 있다면, 이 분류 문제에 대한 상식 수준의 접근법은 새로운 샘플을 항상 클래스 'A'라고 예측하는 것입니다. 이 분류기는 전반적으로 90%의 정확도를 낼 것입니다. 머신 러닝 기반의 모델이라면 90% 이상을 달성해야 유용하다고 볼 수 있습니다. 이따금 이런 기본적인 기준점을 넘어서기가 아주 어려운 경우가 있습니다.

이 경우 온도 시계열 데이터는 연속성이 있고 일자별로 주기성을 가진다고 가정할 수 있습니다(오늘 온도는 내일 온도와 비슷할 가능성이 높습니다). 그렇기 때문에 상식 수준의 해결책은 지금으로부터 24시간 후 온도는 지금과 동일하다고 예측하는 것입니다. 이 방법을 평균 절댓값 오차(MAE)로 평가해 보겠습니다.

```
np.mean(np.abs(preds - targets))
```

다음은 평가를 위한 반복 루프입니다.

코드 6-35 상식적인 기준 모델의 MAE 계산하기

```
def evaluate_naive_method():
    batch_maes = []
    for step in range(val_steps):
        samples, targets = next(val_gen)
        preds = samples[:, -1, 1]
        mae = np.mean(np.abs(preds - targets))
        batch_maes.append(mae)
    print(np.mean(batch_maes))

evaluate_naive_method()
```

출력된 MAE는 0.29입니다. 이 온도 데이터는 평균이 0이고 표준 편차가 1이므로 결괏값이 바로 와닿지는 않습니다. 평균 절댓값 오차 0.29에 표준 편차를 곱하면 섭씨 2.57°C가 됩니다.

코드 6-36 MAE를 섭씨 단위로 변환하기[39]

```
celsius_mae = 0.29 * std[1]
```

평균 절댓값 오차가 상당히 크네요. 이제 딥러닝 모델이 더 나은지 시도해 봅시다.

39 옮긴 코드 6-32에서 계산한 std는 14개의 특성에 대한 표준 편차가 모두 기록된 넘파이 배열입니다. 온도(T (degC))는 인덱스 1입니다.

6.3.4 기본적인 머신 러닝 방법

머신 러닝 모델을 시도하기 전에 상식 수준의 기준점을 세워 놓았습니다. 비슷하게 RNN처럼 복잡하고 연산 비용이 많이 드는 모델을 시도하기 전에 간단하고 손쉽게 만들 수 있는 머신 러닝 모델(예를 들어 소규모의 완전 연결 네트워크)을 먼저 만드는 것이 좋습니다. 이를 바탕으로 더 복잡한 방법을 도입하는 근거가 마련되고 실제적인 이득도 얻게 될 것입니다.

다음 코드는 데이터를 펼쳐서 2개의 Dense 층을 통과시키는 완전 연결 네트워크를 보여 줍니다. 전형적인 회귀 문제이므로 마지막 Dense 층에 활성화 함수를 두지 않았습니다. 손실 함수는 MAE입니다. 상식 수준의 방법에서 사용한 것과 동일한 데이터와 지표를 사용했으므로 결과를 바로 비교해 볼 수 있습니다.

코드 6-37 완전 연결 모델을 훈련하고 평가하기

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Flatten(input_shape=(lookback // step, float_data.shape[-1])))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(1))
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=20,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

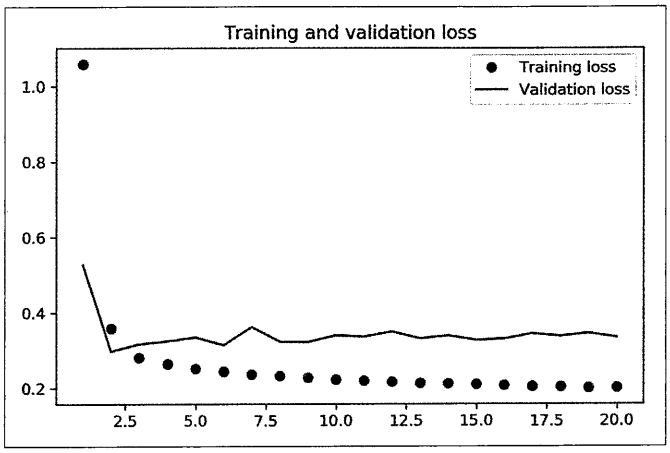
훈련 손실과 검증 손실의 그래프를 그려 보죠(그림 6-20 참고).

코드 6-38 결과 그래프 그리기

```
import matplotlib.pyplot as plt

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)

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()
```

❮ 그림 6-20 간단한 완전 연결 네트워크에서 예나 온도 예측 문제의 훈련 손실과 검증 손실



일부 검증 손실은 학습을 사용하지 않은 기준점에 가깝지만 안정적이지 못합니다. 앞서 기준 모델을 만든 것이 도움이 됩니다. 이 문제는 기준 모델의 성능을 앞지르기가 쉽지 않습니다. 우리가 적용한 상식에는 머신 러닝 모델이 찾지 못한 핵심 정보가 많이 포함되어 있습니다.

간단하고 괜찮은 성능을 내는 모델(상식 수준의 기준 모델)이 데이터와 타깃을 매핑할 수 있다면 왜 훈련한 모델은 이를 찾지 못하고 성능이 낮을까요? 훈련 과정이 찾는 것은 이런 간단한 모델이 아니기 때문입니다. 문제 해결을 위해 탐색하는 모델의 공간, 즉 가설 공간은 우리가 매개변수로 설정한 2개의 층을 가진 네트워크의 모든 가능한 가중치 조합입니다. 이 네트워크는 이미 매우 복잡합니다. 복잡한 모델 공간에서 해결책을 탐색할 때 간단하고 괜찮은 성능을 내는 모델은 찾지 못할 수 있습니다. 심지어 기술적으로 보았을 때, 이 가설 공간에 포함되어 있을 때조차도 말이죠. 이것이 일반적으로 머신 러닝이 가진 심각한 제약 사항입니다. 학습 알고리즘이 특정한 종류의 간단한 모델을 찾도록 하드코딩되지 않았다면, 모델 파라미터를 학습하는 방법은 간단한 문제를 위한 간략한 해결책을 찾지 못할 수 있습니다.

6.3.5 첫 번째 순환 신경망

첫 번째 완전 연결 네트워크는 잘 작동하지 않았습니다. 그렇다고 이 문제에 머신 러닝이 적합하지 않다는 뜻은 아닙니다. 앞선 모델은 시계열 데이터를 펼쳤기 때문에 입력 데이터에서 시간 개념을 잃어버렸습니다. 그 대신 인과 관계와 순서가 의미 있는 시퀀스 데이터를 그대로 사용해 보겠습니다. 이런 시퀀스 데이터에 아주 잘 들어맞는 순환 시퀀스 처리 모델을 시도해 보겠습니다. 이 모델은 앞선 모델과 달리 데이터 포인터의 시간 순서를 사용합니다.

이전 절에서 소개한 LSTM 층 대신에 2014년에 정준영 등이 개발한 GRU 층을 사용하겠습니다.[40] GRU(Gated Recurrent Unit) 층은 LSTM과 같은 원리로 작동하지만 조금 더 간결하고, 그래서 계산 비용이 덜 듭니다(LSTM만큼 표현 학습 능력이 높지는 않을 수 있습니다).[41] 계산 비용과 표현 학습 능력 사이의 트레이드오프(trade-off)는 머신 러닝 어디에서나 등장합니다.

코드 6-39 GRU를 사용한 모델을 훈련하고 평가하기

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.GRU(32, input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=20,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

40 Junyoung Chung et al., "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling," Conference on Neural Information Processing Systems (2014), https://arxiv.org/abs/1412.3555.

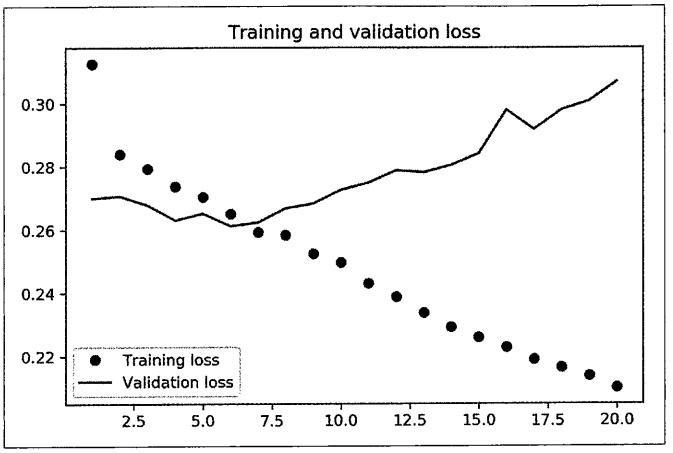
41 **역주** GRU 셀의 상태는 하나고, 하나의 게이트가 삭제 게이트와 입력 게이트의 역할을 합니다. GRU 셀의 의사코드를 코드 6-25와 같은 형태로 표현하면 다음과 같습니다.

```
output_t = z_t x state_t + (1 - z_t) x g_t

z_t = sigmoid(dot(state_t, Uz) + dot(input_t, Wz) + bz)
r_t = sigmoid(dot(state_t, Ur) + dot(input_t, Wr) + br)
g_t = tanh(dot(r_t x state_t, Ug) + dot(input_t, Wg) + bg)
```

GRU 셀에 대한 좀 더 자세한 설명은 〈핸즈온 머신러닝〉(한빛미디어, 2018)의 14장을 참고하세요.

그림 6-21이 결과를 보여 줍니다. 훨씬 좋군요! 상식 수준의 모델을 크게 앞질렀습니다. 이 그림은 시퀀스를 펼쳐서 처리하는 완전 연결 네트워크에 비해서 순환 네트워크가 이런 종류의 작업에 훨씬 뛰어나다는 것과 머신 러닝의 가치를 보여 줍니다.

❮ 그림 6-21 GRU를 사용한 예나 온도 예측 작업의 훈련 손실과 검증 손실



새로운 검증 MAE는 0.265 이하(크게 과대적합되기 시작하는 곳)이고 정규화되기 전인 섭씨로 복원하면 MAE는 2.35℃입니다. 초기 에러 2.57℃보다는 확실히 낮지만 더 개선할 수 있을 것 같습니다.

6.3.6 과대적합을 감소하기 위해 순환 드롭아웃 사용하기

훈련 손실과 검증 손실 곡선을 보면 모델이 과대적합인지 알 수 있습니다. 몇 번의 에포크 이후에 훈련 손실과 검증 손실이 현저하게 벌어지기 시작합니다. 이런 현상을 해결하기 위해 잘 알려진 드롭아웃 기법을 이미 보았습니다. 훈련 데이터를 층에 주입할 때 데이터에 있는 우연한 상관관계를 깨뜨리기 위해 입력 층의 유닛을 랜덤하게 끄는 기법입니다. 순환 신경망에 드롭아웃을 올바르게 적용하는 방법은 간단하지 않습니다. 순환 층 이전에 드롭아웃을 적용하면 규제에 도움이 되는 것보다 학습에 더 방해되는 것으로 오랫동안 알려졌습니다. 2015년 야린 갈(Yarin Gal)이 베이지안 딥러닝에 관한 박사 논문[42]에서 순환 네트워크에 적절하게 드롭아웃을 사용하는 방법을 알아냈

42 Yarin Gal, "Uncertainty in Deep Learning (PhD Thesis)," October 13, 2016, http://mlg.eng.cam.ac.uk/yarin/blog_2248.html.

습니다. 타임스텝마다 랜덤하게 드롭아웃 마스크를 바꾸는 것이 아니라 동일한 드롭아웃 마스크(동일한 유닛의 드롭 패턴)를 모든 타임스텝에 적용해야 합니다. GRU나 LSTM 같은 순환 게이트에 의해 만들어지는 표현을 규제하려면 순환 층 내부 계산에 사용된 활성화 함수에 타임스텝마다 동일한 드롭아웃 마스크를 적용해야 합니다(순환 드롭 아웃 마스크). 모든 타임스텝에 동일한 드롭아웃 마스크를 적용하면 네트워크가 학습 오차를 타임스텝에 걸쳐 적절하게 전파시킬 것입니다. 타임스텝마다 랜덤한 드롭아웃 마스크를 적용하면 오차 신호가 전파되는 것을 방해하고 학습 과정에 해를 끼칩니다.

야린 갈은 케라스를 사용하여 연구했고 케라스 순환 층에 이 기능을 구현하는 데 도움을 주었습니다. 케라스에 있는 모든 순환 층은 2개의 드롭아웃 매개변수를 가지고 있습니다. dropout은 층의 입력에 대한 드롭아웃 비율을 정하는 부동 소수 값입니다. recurrent_dropout은 순환 상태의 드롭아웃 비율을 정합니다.[43] GRU 층에 드롭아웃과 순환 드롭아웃을 적용하여 과대적합에 어떤 영향을 미치는지 살펴보겠습니다. 드롭아웃으로 규제된 네트워크는 언제나 완전히 수렴하는 데 더 오래 걸립니다. 에포크를 2배 더 늘려 네트워크를 훈련하겠습니다.

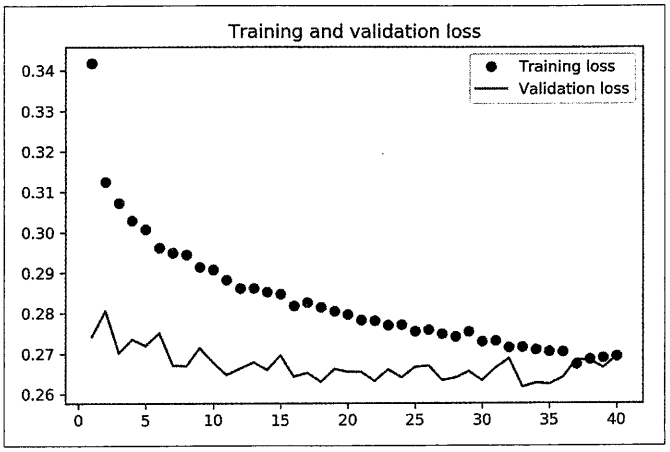
코드 6-40 드롭아웃 규제된 GRU를 사용한 모델을 훈련하고 평가하기

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.GRU(32,
                     dropout=0.2,
                     recurrent_dropout=0.2,
                     input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=40,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

43 옮긴 LSTM 셀의 경우 코드 6-25의 input_t에 dropout 비율이 적용되고 state_t에 recurrent_dropout 비율이 적용됩니다. 주의할 점은 코드 6-25에 사용되는 4개의 input_t에 각기 다른 드롭아웃 마스크가 적용된다는 것입니다. 마찬가지로 4개의 state_t에도 각기 다른 드롭아웃 마스크가 적용됩니다. 주석 41의 GRU 셀의 경우 z_t, r_t, g_t 계산에 사용되는 3개의 input_t와 state_t에도 각기 다른 드롭아웃 마스크가 적용됩니다. output_t 계산에 사용되는 state_t에는 드롭아웃 마스크를 적용하지 않습니다.

그림 6-22에 결과가 나타나 있습니다. 성공이네요! 30번째 에포크까지 과대적합이 일어나지 않았습니다. 평가 점수는 안정적이지만 이전보다 더 나아지지는 않았습니다.

❮ 그림 6-22 드롭아웃 규제된 GRU를 사용한 예나 온도 예측 문제의 훈련 손실과 검증 손실



6.3.7 스태킹 순환 층

과대적합은 더 이상 없지만 성능상 병목이 있는 것 같으므로 네트워크의 용량을 늘려야 합니다. 일반적인 머신 러닝 작업 흐름을 기억하세요. (드롭아웃 등을 사용하여 과대적합을 줄이는 기본 단계를 거쳤다 가정하고) 과대적합이 일어날 때까지 네트워크의 용량을 늘리는 것이 좋습니다. 너무 많이 과대적합되지 않는 한 아직 충분한 용량에 도달한 것이 아닙니다.

네트워크의 용량을 늘리려면 일반적으로 층에 있는 유닛의 수를 늘리거나 층을 더 많이 추가합니다. 순환 층 스태킹은 더 강력한 순환 네트워크를 만드는 고전적인 방법입니다. 예를 들어 구글 번역 알고리즘의 현재 성능은 7개의 대규모 LSTM 층을 쌓은 대규모 모델에서 나온 것입니다.[44]

44 역주 2016년에 나온 한 논문(https://arxiv.org/abs/1609.08144)에 구글의 기계 번역 시스템이 잘 소개되어 있습니다. 여기에서는 양방향 RNN 층을 각기 나누어 8개의 LSTM 층으로 설명했습니다.

케라스에서 순환 층을 차례대로 쌓으려면 모든 중간층은 마지막 타임스텝 출력만 아니고 전체 시퀀스(3D 텐서)를 출력해야 합니다. return_sequences=True로 지정하면 됩니다.

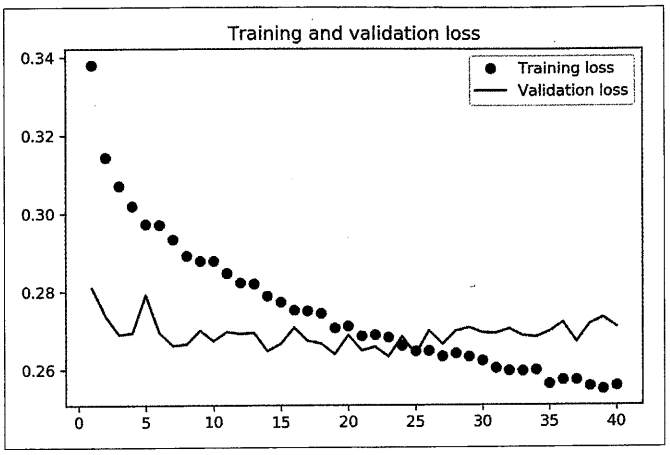
코드 6-41 드롭아웃으로 규제하고 스태킹한 GRU 모델을 훈련하고 평가하기

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.GRU(32,
                     dropout=0.1,
                     recurrent_dropout=0.5,
                     return_sequences=True,
                     input_shape=(None, float_data.shape[-1])))
model.add(layers.GRU(64, activation='relu',
                     dropout=0.1,
                     recurrent_dropout=0.5))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=40,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

그림 6-23이 결과를 보여 줍니다. 층을 추가하여 성능을 조금 향상시켰지만 크지는 않습니다. 여기서 두 가지 결론을 얻을 수 있습니다.

- 아직 충분히 과대적합을 만들지 못했기 때문에 검증 손실을 향상하기 위해서 층의 크기를 늘릴 수 있습니다. 하지만 적지 않은 계산 비용이 추가됩니다.
- 층을 추가한 만큼 도움이 되지 않았으므로, 여기서는 네트워크의 용량을 늘리는 것이 도움이 되지 않는다고 볼 수 있습니다.

❤ 그림 6-23 스태킹 GRU 네트워크를 사용한 예나 온도 예측 문제의 훈련 손실과 검증 손실



6.3.8 양방향 RNN 사용하기

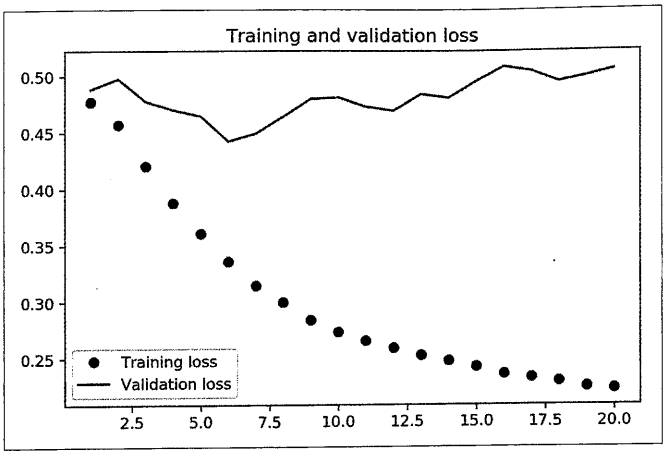
이 절에서 소개할 마지막 기법은 **양방향 RNN**(bidirectional RNN)입니다. 양방향 RNN은 RNN의 한 변종이고 특정 작업에서 기본 RNN보다 훨씬 좋은 성능을 냅니다. 자연어 처리에서는 맥가이버 칼이라고 할 정도로 즐겨 사용됩니다.

RNN은 특히 순서 또는 시간에 민감합니다. 즉 입력 시퀀스의 타임스텝 순서대로 처리합니다. 타임스텝을 섞거나 거꾸로 하면 RNN이 시퀀스에서 학습하는 표현을 완전히 바꾸어 버립니다. 이는 온도 예측처럼 순서에 의미가 있는 문제에 잘 맞는 이유이기도 합니다. 양방향 RNN은 RNN이 순서에 민감하다는 성질을 사용합니다. 앞서 보았던 `GRU`나 `LSTM` 같은 RNN 2개를 사용합니다. 각 RNN은 입력 시퀀스를 한 방향(시간 순서나 반대 순서)으로 처리한 후 각 표현을 합칩니다. 시퀀스를 양쪽 방향으로 처리하기 때문에 양방향 RNN은 단방향 RNN이 놓치기 쉬운 패턴을 감지할 수 있습니다.

놀랍게도 이 절에 있는 RNN 층이 시간의 순서대로 (오래된 타임스텝이 먼저 나오도록) 시퀀스를 처리하는 것은 근거 없는 결정입니다. 적어도 이 결정을 궁금해하지 않았습니다. 시간의 반대 방향으로 (최근 타임스텝이 먼저 나오도록) 입력 시퀀스를 처리하면 만족할 만한 RNN 성능을 낼 수 있을까요? 실제 이렇게 해 보고 결과가 어떤지 확인해 보죠. 해야 할 일은 입력 시퀀스를 시간 차원을 따라 거꾸로 생성하는 데이터 제너레이터를 만드는 것뿐입니다(제너레이터 함수의 마지막

줄을 yield samples[:, ::-1, :], targets로 바꿉니다). 이 절의 첫 번째 예제와 동일하게 하나의 GRU 층을 가진 네트워크를 훈련합니다. 그림 6-24는 훈련 결과입니다.

그림 6-24 거꾸로 뒤집힌 시퀀스로 훈련된 GRU 네트워크를 사용한 예나 온도 예측 문제의 훈련 손실과 검증 손실



순서를 뒤집은 GRU는 상식 수준의 기준점보다도 성능이 낮습니다. 이 경우에는 시간 순서대로 처리하는 것이 중요한 역할을 합니다. 사실 이는 당연합니다. 기본적인 GRU 층은 먼 과거보다 최근 내용을 잘 기억합니다. 또 최근에 가까운 날씨 데이터 포인트일수록 오래된 데이터 포인트보다 예측에 유용합니다(상식 수준의 기준점이 꽤 강력한 이유입니다). 시간 순서대로 처리하는 네트워크가 거꾸로 처리하는 것보다 성능이 높아야만 합니다. 하지만 자연어 처리를 포함하여 다른 많은 문제에서는 그렇지 않습니다. 문장을 이해하는 데 단어의 중요성은 단어가 문장 어디에 놓여 있는지에 따라 결정되지 않습니다. 같은 기법을 6.2절의 LSTM IMDB 예제에 적용해 보죠.

코드 6-42 거꾸로 된 시퀀스를 사용한 LSTM을 훈련하고 평가하기

```
from keras.datasets import imdb
from keras.preprocessing import sequence
from keras import layers
from keras.models import Sequential

max_features = 10000 -------- 특성으로 사용할 단어의 수
maxlen = 500 -------- 사용할 텍스트의 길이(가장 빈번한 max_features개의 단어만 사용합니다.)

(x_train, y_train), (x_test, y_test) = imdb.load_data(
    num_words=max_features) -------- 데이터 로드
```

```
x_train = [x[::-1] for x in x_train]
x_test = [x[::-1] for x in x_test]
```

시퀀스를 뒤집습니다.

```
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
```

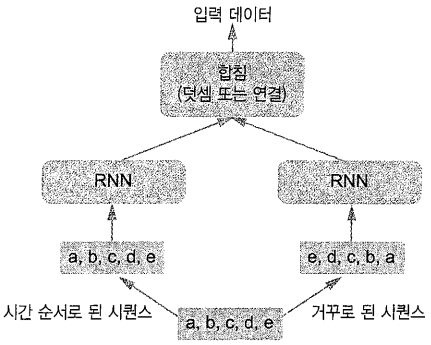
시퀀스에 패딩을 추가합니다.

```
model = Sequential()
model.add(layers.Embedding(max_features, 128))
model.add(layers.LSTM(32))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

시간 순서로 훈련한 LSTM과 거의 동일한 성능을 얻을 수 있습니다. 놀랍게도 이런 텍스트 데이터셋에는 순서를 뒤집어 처리하는 것이 시간 순서대로 처리하는 것과 거의 동일하게 잘 작동합니다. 이는 언어를 이해하는 데 단어의 순서가 중요하지만 결정적이지 않다는 가정을 뒷받침합니다. 거꾸로 된 시퀀스에서 훈련한 RNN은 원래 시퀀스에서 훈련한 것과는 다른 표현을 학습합니다. 이와 비슷하게 시작할 때 죽고 마지막 날 태어나는 삶처럼 실제 세상의 시간이 거꾸로 흘러간다면 우리의 정신 세계는 달라질 것입니다. 머신 러닝에서 다른 표현이 유용하다면 항상 사용할 가치가 있습니다. 이 표현이 많이 다를수록 더 좋습니다. 이 표현이 데이터를 바라보는 새로운 시각을 제공하고 다른 방식에서는 놓칠 수 있는 데이터의 특징을 잡아냅니다. 이런 표현은 작업의 성능을 올리는 데 도움을 줍니다. 이것이 7장에서 살펴볼 앙상블(ensemble) 개념입니다.

양방향 RNN은 이 아이디어를 사용하여 시간 순서대로 처리하는 RNN의 성능을 향상시킵니다. 입력 시퀀스를 양쪽 방향으로 바라보기 때문에(그림 6-25 참고), 드러나지 않은 다양한 표현을 얻어 시간 순서대로 처리할 때 놓칠 수 있는 패턴을 잡아냅니다.

❯ 그림 6-25 양방향 RNN 층의 동작 방식



케라스에서는 Bidirectional 층을 사용하여 양방향 RNN을 만듭니다. 이 클래스는 첫 번째 매개변수로 순환 층의 객체를 전달받습니다. Bidirectional 클래스는 전달받은 순환 층으로 새로운 두 번째 객체를 만듭니다. 하나는 시간 순서대로 입력 시퀀스를 처리하고, 다른 하나는 반대 순서로 입력 시퀀스를 처리합니다. IMDB 감성 분석 문제에 이를 적용해 보죠.

코드 6-43 양방향 LSTM을 훈련하고 평가하기

```
model = Sequential()
model.add(layers.Embedding(max_features, 32))
model.add(layers.Bidirectional(layers.LSTM(32)))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

여기서 얻은 검증 정확도는 88% 정도로 이전 절에서 사용했던 일반 LSTM보다 성능이 조금 더 높습니다. 이 모델은 조금 더 일찍 과대적합되는 것 같습니다. 양방향 순환 층이 단방향 LSTM보다 모델 파라미터가 2배 많기 때문에 놀라운 일은 아닙니다. 규제를 조금 추가한다면 양방향 순환 층을 사용하는 것이 이 작업에 더 적합해 보입니다.

이제 동일한 방식을 온도 예측 문제에 적용해 보죠.

코드 6-44 양방향 GRU 훈련하기

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Bidirectional(
    layers.GRU(32), input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=40,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

이 네트워크는 일반 GRU 층과 비슷한 성능을 냅니다. 이유는 쉽게 이해할 수 있습니다. 모든 예측 성능은 시간 순서대로 처리하는 네트워크의 절반에서 옵니다. 시간 반대 순서로 처리하는 절반은 이런 작업에 성능이 매우 좋지 않기 때문입니다(최근 정보가 오래 전 정보보다 훨씬 더 중요합니다).

6.3.9 더 나아가서

온도 예측 문제의 성능을 향상하기 위해 시도해 볼 수 있는 것들이 많이 있습니다.

- 스태킹한 각 순환 층의 유닛 수를 조정합니다. 지금 설정은 대부분 임의로 한 것이라 최적화가 덜 되었을 것입니다.
- RMSprop 옵티마이저가 사용한 학습률을 조정합니다.
- GRU 대신 LSTM 층을 사용합니다.
- 순환 층 위에 용량이 큰 완전 연결된 회귀 층을 사용합니다. 즉 유닛 수가 많은 Dense 층 한 개 또는 여러 개의 Dense 층을 스태킹합니다
- 최종적으로 (검증 MAE 기준으로 보았을 때) 최선의 모델을 테스트 세트에서 확인해야 합니다. 이를 잊으면 검증 세트에 과대적합된 네트워크 구조를 만들게 될 것입니다.

늘 그렇듯이 딥러닝은 과학보다는 예술에 가깝습니다. 어떤 문제에 적합하거나 그렇지 않은 가이드라인은 제시할 수 있지만 결국 모든 문제는 다릅니다. 경험을 바탕으로 다른 전략들을 시도해 보아야 합니다. 현재는 문제를 해결하는 최선의 방법을 미리 알 수 있는 이론이 없습니다. 반복해서 시도해야 합니다.

6.3.10 정리

다음은 이 절에서 배운 것들입니다.

- 4장에서 처음 배웠던 것처럼 새로운 문제를 해결할 때는 선택한 지표에서 상식 수준의 기준점을 설정하는 것이 좋습니다. 기준점을 가지고 있지 않으면 실제로 향상되었는지 알 수 없습니다.
- 계산 비용을 추가할지 판단하기 위해서 비용이 비싼 모델 전에 간단한 모델을 시도합니다.
- 시간 순서가 중요한 데이터가 있다면 순환 층이 적합합니다. 시계열 데이터를 펼쳐서 처리하는 모델의 성능을 쉽게 앞지를 것입니다.
- 순환 네트워크에 드롭아웃을 사용하려면 타임스텝 동안 일정한 드롭아웃 마스크와 순환 드롭아웃 마스크를 사용해야 합니다. 둘 다 케라스 순환 층에 포함되어 있습니다. 순환 층에 있는 `dropout`과 `recurrent_dropout` 매개변수를 사용하면 됩니다.
- 스태킹 RNN은 단일 RNN 층보다 더 강력한 표현 능력을 제공합니다. 하지만 계산 비용이 많이 들기 때문에 항상 시도할 가치가 있지는 않습니다. (기계 번역 같은) 복잡한 문제에서 확실히 도움이 되지만 작고 간단한 문제에서는 항상 그렇지 않습니다.
- 양쪽 방향으로 시퀀스를 바라보는 양방향 RNN은 자연어 처리 문제에 유용합니다. 하지만 최근 정보가 오래된 것보다 훨씬 의미 있는 시퀀스 데이터에는 잘 작동하지 않습니다.

Note ≡ 여기서 자세히 다루지 않은 두 가지 중요한 개념이 있습니다. 순환 어텐션(attention)과 시퀀스 마스킹(sequence masking)입니다. 둘 다 자연어 처리에 깊게 관련되어 있고 온도 예측 문제에는 적합하지 않습니다. 이 책을 끝내고 앞으로 공부할 목록으로 남겨 두겠습니다.

Note ≡ 주식 시장과 머신 러닝

일부 독자들은 여기서 소개한 기법을 주식 시장의 증권 가격(또는 환율 등)을 예측하는 데 사용하려고 할 것입니다. 주식 시장은 날씨 패턴 같은 자연 현상과는 훨씬 다른 통계적 특성이 있습니다. 공개된 데이터만 가지고 머신 러닝을 주식 가격 문제에 적용하기는 매우 어렵습니다. 아마도 시간과 자원을 낭비하고 아무것도 얻지 못할 것입니다.

주식 시장에서 과거 성과는 미래의 기대 수익을 위한 좋은 예측 특성이 아님을 항상 기억하세요. 마치 백미러를 보고 운전하는 것과 같습니다. 머신 러닝을 적용할 수 있는 곳은 과거를 미래에 대한 좋은 예측 지표로 쓸 수 있는 데이터셋입니다.

DEEP LEARNING

6.4 컨브넷을 사용한 시퀀스 처리

5장에서 합성곱 신경망(컨브넷)이 무엇인지 그리고 컴퓨터 비전 문제에 어떻게 잘 맞는지 배웠습니다. 입력의 부분 패치에서 특성을 뽑아내어 구조적인 표현을 만들고 데이터를 효율적으로 사용하는 합성곱 연산의 능력 때문입니다. 컴퓨터 비전에서 뛰어난 컨브넷의 특징이 시퀀스 처리와도 깊게 관련되어 있습니다. 시간을 2D 이미지의 높이와 너비 같은 공간의 차원으로 다룰 수 있습니다.

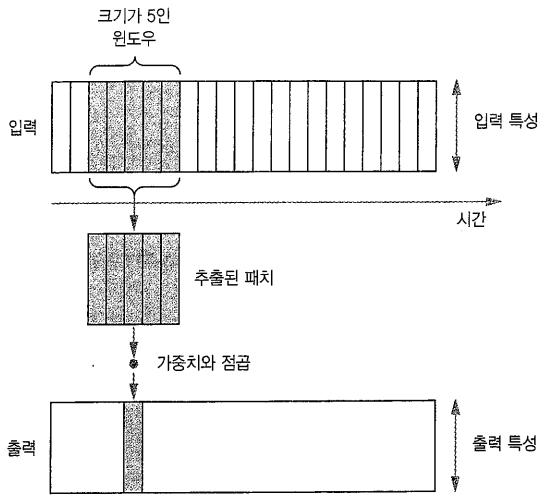
1D 컨브넷(1D Convnet)은 특정 시퀀스 처리 문제에서 RNN과 견줄 만합니다. 일반적으로 계산 비용이 훨씬 쌉니다. 1D 컨브넷은 전형적으로 팽창된 커널(dilated kernel)과 함께 사용됩니다.[45] 최근에 오디오 생성과 기계 번역 분야에서 큰 성공을 거두었습니다. 이런 특정 분야의 성공 이외에도 텍스트 분류나 시계열 예측 같은 간단한 문제에서 작은 1D 컨브넷이 RNN을 대신하여 빠르게 처리할 수 있다고 알려져 있습니다.

45 역주 팽창 합성곱(dilated convolution)은 커널에 구멍을 추가하여 입력을 건너뛰면서 합성곱하는 것과 같습니다. 프랑스어로 '구멍이 난'이란 뜻의 아트루스(àtrous)에서 따와 아트루스 합성곱(atrous convolution)이라고도 부릅니다. 케라스에서는 `Conv1D`와 `Conv2D`의 `dilation_rate` 매개변수에서 팽창 비율을 지정할 수 있습니다. 팽창 합성곱은 이미지 분할(image segmentation) 분야 등에 많이 사용됩니다. 구글 딥마인드가 WaveNet(https://arxiv.org/abs/1609.03499)에 적용하여 많이 알려졌습니다.

6.4.1 시퀀스 데이터를 위한 1D 합성곱 이해하기

앞서 소개한 합성곱 층은 2D 합성곱입니다. 이미지 텐서에서 2D 패치를 추출하고 모든 패치에 동일한 변환을 적용합니다. 같은 방식으로 시퀀스에서 1D 패치(부분 시퀀스)를 추출하여 1D 합성곱을 적용합니다(그림 6-26 참고).

❤ **그림 6-26** 1D 합성곱 작동 방식: 입력 시퀀스에서 시간 축으로 패치를 추출하여 출력 타임스텝을 만든다



이런 1D 합성곱 층은 시퀀스에 있는 지역 패턴을 인식할 수 있습니다. 동일한 변환이 시퀀스에 있는 모든 패치에 적용되기 때문에 특정 위치에서 학습한 패턴을 나중에 다른 위치에서 인식할 수 있습니다. 이는 1D 컨브넷에 (시간의 이동에 대한) 이동 불변성(translation invariant)을 제공합니다. 예를 들어 크기 5인 윈도우를 사용하여 문자 시퀀스를 처리하는 1D 컨브넷은 5개 이하의 단어나 단어의 부분을 학습합니다. 이 컨브넷은 이 단어가 입력 시퀀스의 어느 문장에 있더라도 인식할 수 있습니다. 따라서 문자 수준의 1D 컨브넷은 단어 형태학(word morphology)에 관해 학습할 수 있습니다.[46]

46 참고 2016년에 나온 논문 Jason Lee et al., "Fully Character-Level Neural Machine Translation without Explicit Segmentation"(https://arxiv.org/abs/1610.03017)을 말합니다.

6.4.2 시퀀스 데이터를 위한 1D 풀링

컨브넷에서 이미지 텐서의 크기를 다운샘플링하기 위해 사용하는 평균 풀링이나 맥스 풀링 같은 2D 풀링 연산을 배웠습니다. 1D 풀링 연산은 2D 풀링 연산과 동일합니다. 입력에서 1D 패치(부분 시퀀스)를 추출하고 최댓값(최대 풀링)을 출력하거나 평균값(평균 풀링)을 출력합니다. 2D 컨브넷과 마찬가지로 1D 입력의 길이를 줄이기 위해 사용합니다(**서브샘플링**(subsampling)).

6.4.3 1D 컨브넷 구현

케라스에서 1D 컨브넷은 `Conv1D` 층을 사용하여 구현합니다. `Conv1D`는 `Conv2D`와 인터페이스가 비슷합니다. `(samples, time, features)` 크기의 3D 텐서를 입력받고 비슷한 형태의 3D 텐서를 반환합니다. 합성곱 윈도우는 시간 축의 1D 윈도우입니다. 즉 입력 텐서의 두 번째 축입니다.

간단한 2개의 층으로 된 1D 컨브넷을 만들어 익숙한 IMDB 감성 분류 문제에 적용해 보죠. 기억을 되살리기 위해 데이터를 로드하고 전처리하는 코드를 다시 보겠습니다.

코드 6-45 IMDB 데이터 전처리하기

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

max_features = 10000
max_len = 500

print('데이터 로드...')
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
print(len(x_train), '훈련 시퀀스')
print(len(x_test), '테스트 시퀀스')

print('시퀀스 패딩 (samples x time)')
x_train = sequence.pad_sequences(x_train, maxlen=max_len)
x_test = sequence.pad_sequences(x_test, maxlen=max_len)
print('x_train 크기:', x_train.shape)
print('x_test 크기:', x_test.shape)
```

1D 컨브넷은 5장에서 사용한 2D 컨브넷과 비슷한 방식으로 구성합니다. Conv1D와 MaxPooling1D 층을 쌓고 **전역 풀링 층**[47]이나 Flatten 층으로 마칩니다. 이 구조는 3D 입력을 2D 출력으로 바꾸므로 분류나 회귀를 위해 모델에 하나 이상의 Dense 층을 추가할 수 있습니다.

한 가지 다른 점은 1D 컨브넷에 큰 합성곱 윈도우를 사용할 수 있다는 것입니다. 2D 합성곱 층에서 3×3 합성곱 윈도우는 3 × 3 = 9 특성을 고려합니다. 1D 합성곱 층에서 크기 3인 합성곱 윈도우는 3개의 특성만 고려합니다. 그래서 1D 합성곱에 크기 7이나 9의 윈도우를 사용할 수 있습니다.

다음은 IMDB 데이터셋을 위한 1D 컨브넷의 예입니다.

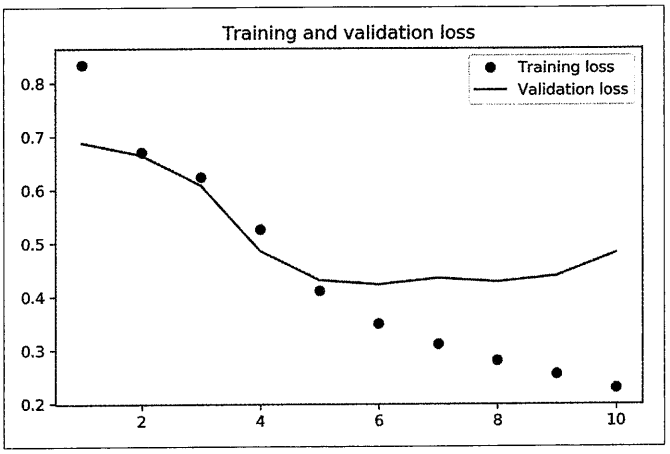
코드 6-46 IMDB 데이터에 1D 컨브넷을 훈련하고 평가하기

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Embedding(max_features, 128, input_length=max_len))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))

model.summary()

model.compile(optimizer=RMSprop(lr=1e-4),
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

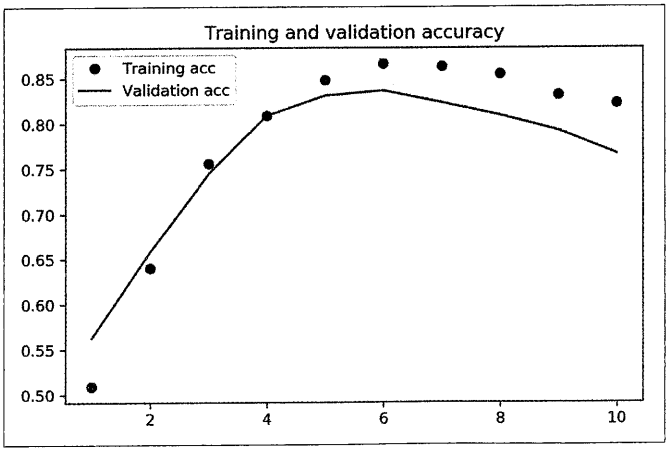
47 옮긴 GlobalAveragePooling1D, GlobalMaxPooling1D 풀링은 (samples, timesteps, features) 크기의 텐서를 입력받고 (samples, features) 크기의 텐서를 출력합니다. 즉 시간 축 전체에 풀링을 적용합니다. GlobalAveragePooling2D, GlobalMaxPooling2D 풀링은 (samples, height, width, channels) 크기의 텐서를 입력받고 (samples, channels) 크기의 텐서를 출력합니다. 즉 특성 맵의 공간 차원 전체에 대한 풀링입니다.

그림 6-27과 그림 6-28은 훈련과 검증 결과를 보여 줍니다. 검증 정확도는 LSTM보다 조금 낮지만 CPU나 GPU에서 더 빠르게 실행됩니다(속도 향상은 환경에 따라 많이 다릅니다). 여기에서 적절한 에포크 수(4개)로 모델을 다시 훈련하고 테스트 세트에서 확인할 수 있습니다. 이 예는 단어 수준의 감성 분류 작업에 순환 네트워크를 대신하여 빠르고 경제적인 1D 컨브넷을 사용할 수 있음을 보여 줍니다.

❮ 그림 6-27 1D 컨브넷을 사용한 IMDB 문제의 훈련 손실과 검증 손실



❮ 그림 6-28 1D 컨브넷을 사용한 IMDB 문제의 훈련 정확도와 검증 정확도



6.4.4 CNN과 RNN을 연결하여 긴 시퀀스를 처리하기

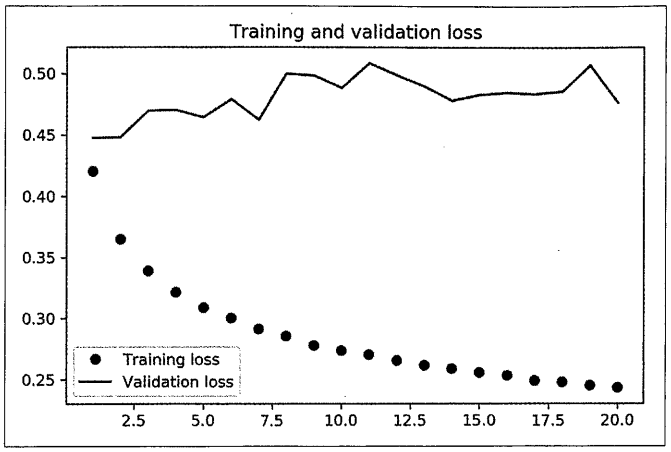
1D 컨브넷이 입력 패치를 독립적으로 처리하기 때문에 RNN과 달리 (합성곱 윈도우 크기의 범위를 넘어서는) 타임스텝의 순서에 민감하지 않습니다. 물론 장기간 패턴을 인식하기 위해 많은 합성곱 층과 풀링 층을 쌓을 수 있습니다. 상위 층은 원본 입력에서 긴 범위를 보게 될 것입니다. 이런 방법은 순서를 감지하기에 부족합니다. 온도 예측 문제에 1D 컨브넷을 적용해서 이를 확인해 보겠습니다. 이 문제는 순서를 감지해야 좋은 예측을 만들어 낼 수 있습니다. 다음 예는 이전에 정의한 `float_data`, `train_gen`, `val_gen`, `val_steps`를 다시 사용합니다.

코드 6-47 예나 데이터에서 1D 컨브넷을 훈련하고 평가하기

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Conv1D(32, 5, activation='relu',
                        input_shape=(None, float_data.shape[-1])))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=20,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

그림 6-29는 훈련 MAE와 검증 MAE를 보여 줍니다.

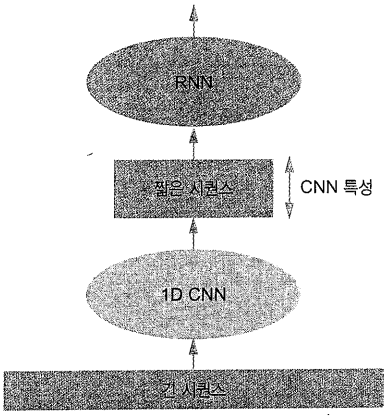
❯ 그림 6-29 1D 컨브넷을 사용한 예나 온도 예측 문제의 훈련 손실과 검증 손실



검증 MAE는 0.40대에 머물러 있습니다. 작은 컨브넷을 사용해서 상식 수준의 기준점을 넘지 못했습니다. 이는 컨브넷이 입력 시계열에 있는 패턴을 보고 이 패턴의 시간 축의 위치(시작인지 끝 부분인지 등)를 고려하지 않기 때문입니다. 최근 데이터 포인트일수록 오래된 데이터 포인트와는 다르게 해석해야 하기 때문에 컨브넷이 의미 있는 결과를 만들지 못합니다. 이런 컨브넷의 한계는 IMDB 데이터에서는 문제가 되지 않습니다. 긍정 또는 부정적인 감성과 연관된 키워드 패턴의 중요성은 입력 시퀀스에 나타난 위치와 무관하기 때문입니다.

컨브넷의 속도와 경량함을 RNN의 순서 감지 능력과 결합하는 한 가지 전략은 1D 컨브넷을 RNN 이전에 전처리 단계로 사용하는 것입니다(그림 6-30 참고). 수천 개의 스텝을 가진 시퀀스 같이 RNN으로 처리하기에는 현실적으로 너무 긴 시퀀스를 다룰 때 특별히 도움이 됩니다. 컨브넷이 긴 입력 시퀀스를 더 짧은 고수준 특성의 (다운샘플된) 시퀀스로 변환합니다. 추출된 특성의 시퀀스는 RNN 파트의 입력이 됩니다.

❤ 그림 6-30 긴 시퀀스를 처리하기 위해 1D 컨브넷과 RNN 결합하기



이 기법이 연구 논문이나 실전 애플리케이션에 자주 등장하지는 않습니다. 아마도 널리 알려지지 않았기 때문일 것입니다. 이 방법은 효과적이므로 많이 사용되기를 바랍니다. 온도 예측 문제에 적용해 보죠. 이 전략은 훨씬 긴 시퀀스를 다룰 수 있으므로 더 오래 전 데이터를 바라보거나(데이터 제너레이터의 lookback 매개변수를 증가시킵니다), 시계열 데이터를 더 촘촘히 바라볼 수 있습니다(제너레이터의 step 매개변수를 감소시킵니다). 여기서는 그냥 step을 절반으로 줄여서 사용하겠습니다. 온도 데이터가 30분마다 1포인트씩 샘플링되기 때문에 결과 시계열 데이터는 2배로 길어집니다. 앞서 정의한 제너레이터 함수를 다시 사용합니다.

코드 6-48 고밀도 데이터 제너레이터로 예나 데이터셋 준비하기

```
step = 3 -------- 이전에는 6이었습니다(시간마다 1 포인트). 이제는 3입니다(30분마다 1포인트).
lookback = 1440 ┐
delay = 144     ┘--- 변경되지 않았습니다!

train_gen = generator(float_data,
                      lookback=lookback,
                      delay=delay,
                      min_index=0,
                      max_index=200000,
                      shuffle=True,
                      step=step)
val_gen = generator(float_data,
                    lookback=lookback,
```

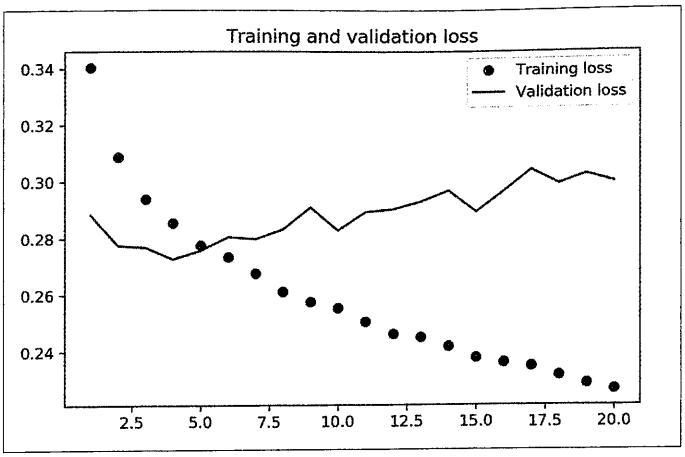
```
                      delay=delay,
                      min_index=200001,
                      max_index=300000,
                      step=step)
test_gen = generator(float_data,
                     lookback=lookback,
                     delay=delay,
                     min_index=300001,
                     max_index=None,
                     step=step)
val_steps = (300000 - 200001 - lookback) // 128
test_steps = (len(float_data) - 300001 - lookback) // 128
```

이 모델은 2개의 Conv1D 층 다음에 GRU 층을 놓았습니다. 그림 6-31이 결과를 보여 줍니다.

코드 6-49 1D 합성곱과 GRU 층을 연결한 모델

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Conv1D(32, 5, activation='relu',
                        input_shape=(None, float_data.shape[-1])))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.GRU(32, dropout=0.1, recurrent_dropout=0.5))
model.add(layers.Dense(1))

model.summary()

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=20,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

❤ 그림 6-31 1D 컨브넷과 GRU를 사용한 예나 온도 예측 문제의 훈련 손실과 검증 손실



검증 손실로 비교해 보면 이 설정은 규제가 있는 GRU 모델만큼 좋지는 않습니다. 하지만 훨씬 빠르기 때문에 데이터를 2배 더 많이 처리할 수 있습니다. 여기에서는 도움이 많이 되지 않았지만 다른 데이터셋에서는 중요할 수 있습니다.

6.4.5 정리

다음은 이 절에서 배운 것들입니다.

- 2D 컨브넷이 2D 공간의 시각적 패턴을 잘 처리하는 것처럼 1D 컨브넷은 시간에 따른 패턴을 잘 처리합니다. 1D 컨브넷은 특정 자연어 처리 같은 일부 문제에 RNN을 대신할 수 있는 빠른 모델입니다.
- 전형적으로 1D 컨브넷은 컴퓨터 비전 분야의 2D 컨브넷과 비슷하게 구성합니다. `Conv1D` 층과 `Max-Pooling1D` 층을 쌓고 마지막에 전역 풀링 연산이나 `Flatten` 층을 둡니다.
- RNN으로 아주 긴 시퀀스를 처리하려면 계산 비용이 많이 듭니다. 1D 컨브넷은 비용이 적게 듭니다. 따라서 1D 컨브넷을 RNN 이전의 전처리 단계로 사용하는 것은 좋은 생각입니다. 시퀀스 길이를 줄이고 RNN이 처리할 유용한 표현을 추출해 줄 것입니다.

6.5 요약

- 이 장에서는 다음 기법들을 배웠습니다. 이 기법들은 텍스트에서 시계열까지 다양한 시퀀스 데이터셋에 폭넓게 적용할 수 있습니다.
  - 텍스트를 토큰화하는 방법
  - 단어 임베딩과 이를 사용하는 방법
  - 순환 네트워크와 이를 사용하는 방법
  - 더 강력한 시퀀스 처리 모델을 만들기 위해 RNN 층을 스태킹하는 방법과 양방향 RNN을 사용하는 방법
  - 시퀀스를 처리하기 위해 1D 컨브넷을 사용하는 방법
  - 긴 시퀀스를 처리하기 위해 1D 컨브넷과 RNN을 연결하는 방법
- 시계열 데이터를 사용한 회귀(미래 값을 예측), 시계열 분류, 시계열에 있는 이상치 감지, 시퀀스 레이블링(문장에서 이름이나 날짜를 식별하기 등)에 RNN을 사용할 수 있습니다.
- 비슷하게 기계 번역(SliceNet[48] 같은 시퀀스-투-시퀀스 합성곱 모델), 문서 분류, 맞춤법 정정 등에 1D 컨브넷을 사용할 수 있습니다.
- 시퀀스 데이터에서 전반적인 순서가 중요하다면 순환 네트워크를 사용하여 처리하는 것이 좋습니다. 최근의 정보가 오래된 과거보다 더 중요한 시계열 데이터가 전형적인 경우입니다.
- 전반적인 순서가 큰 의미가 없다면 1D 컨브넷이 적어도 동일한 성능을 내면서 비용도 적을 것입니다. 텍스트 데이터가 종종 이에 해당합니다. 문장 처음에 있는 키워드가 마지막에 있는 키워드와 같은 의미를 가집니다.

48 https://arxiv.org/abs/1706.03059를 참고하세요.