

```

45000/45000 [=====] - 4s 92us/sample - loss: 0.3246 - accuracy: 0.8802 - val_loss: 0.3496 - val_accuracy: 0.8722
Epoch 5/25
45000/45000 [=====] - 4s 92us/sample - loss: 0.3042 - accuracy: 0.8874 - val_loss: 0.3383 - val_accuracy: 0.8817
Epoch 6/25
45000/45000 [=====] - 4s 92us/sample - loss: 0.2885 - accuracy: 0.8931 - val_loss: 0.3352 - val_accuracy: 0.8813
Epoch 7/25
45000/45000 [=====] - 4s 94us/sample - loss: 0.2754 - accuracy: 0.8980 - val_loss: 0.3379 - val_accuracy: 0.8757
Epoch 8/25
45000/45000 [=====] - 4s 93us/sample - loss: 0.2622 - accuracy: 0.9022 - val_loss: 0.3327 - val_accuracy: 0.8779
Epoch 9/25
45000/45000 [=====] - 4s 93us/sample - loss: 0.2501 - accuracy: 0.9078 - val_loss: 0.3244 - val_accuracy: 0.8877
Epoch 10/25
45000/45000 [=====] - 4s 92us/sample - loss: 0.2432 - accuracy: 0.9082 - val_loss: 0.3425 - val_accuracy: 0.8802
(이하 생략)

```

학습 출력 결과를 보면 훈련 데이터의 정확도가 점점 증가하는 것에 비해 검증 데이터의 정확도는 일정한 수준으로 유지되는 것을 볼 수 있습니다. 전체 학습 과정을 조망하기 위해 history 변수에 저장된 학습 결과를 시각화해보겠습니다.

**예제 5.24** Fashion MNIST 분류 모델의 학습 결과 시각화

### [IN]

```

import matplotlib.pyplot as plt
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], 'b-', label='loss')
plt.plot(history.history['val_loss'], 'r--', label='val_loss')
plt.xlabel('Epoch')
plt.legend()

plt.subplot(1, 2, 2)

```

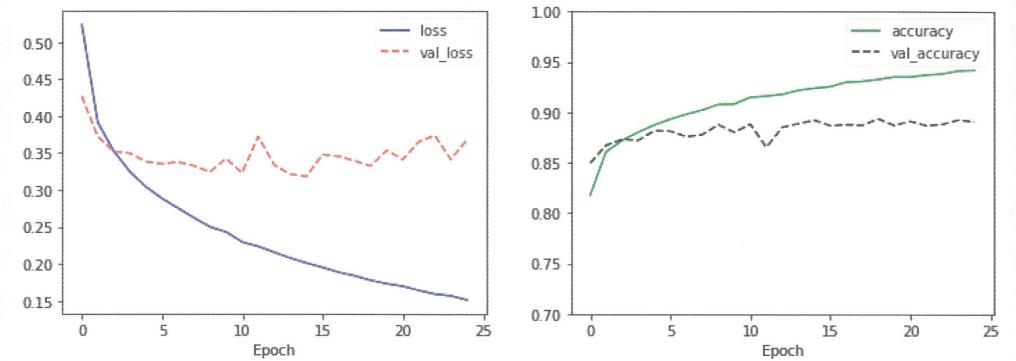
```

plt.plot(history.history['accuracy'], 'g-', label='accuracy')
plt.plot(history.history['val_accuracy'], 'k--', label='val_accuracy')
plt.xlabel('Epoch')
plt.ylim(0.7, 1)
plt.legend()

plt.show()

```

### [OUT]



검증 데이터의 손실이 감소하다가 시간이 지날수록 서서히 증가하는 과적합 현상을 확인할 수 있습니다. 이를 막기 위해서는 예제 4.18에 나왔던 `tf.keras.callbacks.EarlyStopping`을 사용해볼 수 있겠습니다.

**예제 5.25** Fashion MNIST 분류 평가

### [IN]

```

model.evaluate(test_X, test_Y)

```

### [OUT]

```

10000/10000 [=====] - 0s 40us/sample - loss: 0.2132 - accuracy: 0.8850
[0.3982168311655521, 0.885]

```

테스트 데이터에 대한 평가 정확도는 88.5%가 나왔습니다. 괜찮은 수치 같지만 네트워크 구조 변경과 다른 학습 기법을 사용해서 정확도를 90% 이상으로 끌어올릴 수 있습니다. 6장 ‘컨볼루션 신경망(CNN)’에서 그 방법을 알아보겠습니다.

## 5.4 정리

이번 장에서는 회귀와 더불어 머신러닝에서 가장 기초적인 데이터 분석 방법 중 하나인 분류에 대해 알아봤습니다. 분류 문제에서 가장 중요한 두 가지 요소는 소프트맥스 함수와 크로스 엔트로피 계산입니다.

예제로 와인 데이터세트를 활용해 정답이 두 개인 이항 분류 문제와 세 개인 다항 분류 문제를 풀어봤고, 머신러닝의 새로운 벤치마크 데이터세트 중 하나인 Fashion MNIST를 이용해 정답 범주가 10개인 다항 분류 문제를 풀었습니다. 또 원-핫 인코딩에 대해 알아보았고, 케라스에서 희소 행렬을 원-핫 인코딩으로의 명시적 변환 없이 정답 행렬로 사용할 수 있다는 것을 배웠습니다.

## 컨볼루션 신경망

오랜 시간 동안 이미지 데이터는 숫자나 표로 된 데이터에 비해 다루기 어렵다고 여겨졌습니다. 딥러닝과 컨볼루션 신경망(Convolutional Neural Network; CNN)이 대중화되기 전까지는 말입니다.

5장의 첫 부분에서 소개했던 ImageNet 대회의 이미지 분류 문제는 컨볼루션 신경망이 나오기 전까지는 매우 어려운 문제로 생각됐습니다. 5장에 나온 또 다른 문제인 고양이와 개의 사진을 구별하는 문제도 역시 어려운 문제였지만 컨볼루션 신경망은 쉽게 풀 수 있습니다.

최근 컨볼루션 신경망은 이미지뿐 아니라 텍스트나 음성 등 다양한 분야의 데이터 처리에 쓰이고 있습니다. 이번 장에서는 컨볼루션 신경망의 사용법 중 가장 기초가 되는 이미지 데이터를 다루는 방법을 알아봅니다.

### 6.1 특징 추출

4장에 나온 보스턴 주택 가격 데이터세트에는 주택의 가격을 예측하기 위한 주택당 방의 수, 재산세율, 범죄율 같은 특징(feature)들이 있었습니다. 5장에 나온 와인 데이터세트에서도 와인의 종류나 품질을 예측하기 위한 당도, 알코올 도수 같은 특징들이 데이터에 존재했습니다.

이에 비해 Fashion MNIST 같은 이미지 데이터에서는 어떤 특징을 찾을 수 있을까요? 이미지 데이터에서는 연구자가 스스로 특징을 찾아야 합니다. 과거의 비전(Vision) 연구에서는 특징을 찾기 위한 다양한 방법이 개발됐습니다. 이미지에서 사물의 외곽선에 해당하는 특징을 발견하면 물체 감지(object detection) 알고리즘을 만들 수 있습니다. 다른 예로 SIFT(Scale-Invariant Feature Transform) 알고리즘은 이미지의 회전과 크기에 대해 변하지 않는 특징을 추출해서 두 개의 이미지에서 서로 대응되는 부분을 찾아냅니다.

## 컨볼루션 신경망

오랜 시간 동안 이미지 데이터는 숫자나 표로 된 데이터에 비해 다루기 어렵다고 여겨졌습니다. 딥러닝과 컨볼루션 신경망(Convolutional Neural Network; CNN)이 대중화되기 전까지는 말입니다.

5장의 첫 부분에서 소개했던 ImageNet 대회의 이미지 분류 문제는 컨볼루션 신경망이 나오기 전까지는 매우 어려운 문제로 생각됐습니다. 5장에 나온 또 다른 문제인 고양이와 개의 사진을 구별하는 문제도 역시 어려운 문제였지만 컨볼루션 신경망은 쉽게 풀 수 있습니다.

최근 컨볼루션 신경망은 이미지뿐 아니라 텍스트나 음성 등 다양한 분야의 데이터 처리에 쓰이고 있습니다. 이번 장에서는 컨볼루션 신경망의 사용법 중 가장 기초가 되는 이미지 데이터를 다루는 방법을 알아봅니다.

### 6.1 특징 추출

4장에 나온 보스턴 주택 가격 데이터세트에는 주택의 가격을 예측하기 위한 주택당 방의 수, 재산세율, 범죄율 같은 특징(feature)들이 있었습니다. 5장에 나온 와인 데이터세트에서도 와인의 종류나 품질을 예측하기 위한 당도, 알코올 도수 같은 특징들이 데이터에 존재했습니다.

이에 비해 Fashion MNIST 같은 이미지 데이터에서는 어떤 특징을 찾을 수 있을까요? 이미지 데이터에서는 연구자가 스스로 특징을 찾아야 합니다. 과거의 비전(Vision) 연구에서는 특징을 찾기 위한 다양한 방법이 개발됐습니다. 이미지에서 사물의 외곽선에 해당하는 특징을 발견하면 물체 감지(object detection) 알고리즘을 만들 수 있습니다. 다른 예로 SIFT(Scale-Invariant Feature Transform) 알고리즘은 이미지의 회전과 크기에 대해 변하지 않는 특징을 추출해서 두 개의 이미지에서 서로 대응되는 부분을 찾아냅니다.

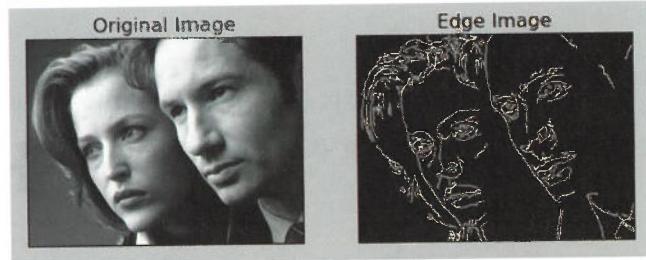


그림 6.1 외곽선 검출 알고리즘 중 하나인 Canny Edge Detection을 이용한 결과 이미지<sup>1</sup>

이런 특징 추출(Feature Extraction) 기법 중 하나인 컨볼루션 연산은 각 픽셀을 본래 픽셀과 그 주변 픽셀의 조합으로 대체하는 동작<sup>2</sup>입니다. 이때 연산에 쓰이는 작은 행렬을 필터(filter) 또는 커널(kernel)이라고 합니다.

컨볼루션 연산은 우리말로 합성곱이라고 합니다. 원본 이미지의 각 픽셀을 포함한 주변 픽셀과 필터의 모든 픽셀은 각각 곱연산을 하고, 그 결과를 모두 합해서 새로운 이미지에 넣어주기 때문에 합성곱이라는 이름이 붙은 것 같습니다.

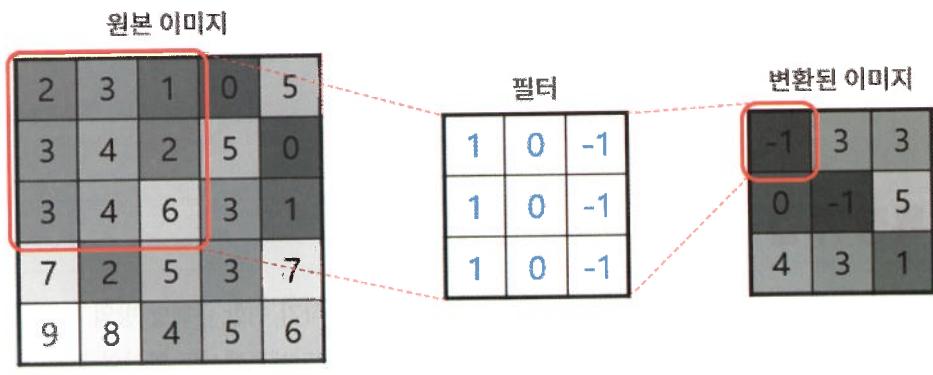


그림 6.2 컨볼루션 연산은 원본 이미지와 필터 행렬의 합성곱입니다.

1 《Canny Edge Detection》, BogoToBogo <http://bit.ly/32niSye>

2 네이버 IT 용어사전: <http://bit.ly/2XIMBTG>

간단한 컨볼루션 필터 몇 가지를 소개해 보겠습니다. 그림 6.3에서 쓰인  $3 \times 3$  크기의 작은 필터는 왼쪽의 원본 이미지를 각각 새로운 이미지로 변환합니다. 이때 각 필터의 생김새에 따라 수직선/수평선 검출, 흐림(blur) 효과, 날카로운(sharpen) 이미지 효과 등을 줄 수 있습니다.



그림 6.3 다양한 필터를 적용했을 때의 컨볼루션 연산의 결과<sup>3</sup>

그런데 이런 필터들은 경험적 지식을 통해 직접 손으로 값을 넣어준 결과입니다. 이것을 수작업으로 설계한 특징(Hand-crafted feature)이라고 합니다. 위에서 본 외곽선 검출 알고리즘이나 SIFT, 그리고 그림 6.3의 다양한 필터는 모두 수작업으로 설계한 특징의 범주에서 벗어나지 않습니다.

그런데 수작업으로 설계한 특징에는 세 가지 문제점이 있습니다.<sup>4</sup> 첫째, 적용하고자 하는 분야에 대한 전문적 지식이 필요합니다. 둘째, 수작업으로 특징을 설계하는 것은 시간과 비용이 많이 드는 작업입니다. 셋째, 한 분야(예를 들어 이미지)에서 효과적인 특징을 다른 분야(예를 들어 음성)에 적용하기 어렵습니다.

딥러닝 기반의 컨볼루션 연산은 이런 문제점을 모두 해결했습니다. 컨볼루션 신경망은 특징을 검출하는 필터를 수작업으로 설계하는 것이 아니라 네트워크가 특징을 추출하는 필터를 자동으로 생성합니다. 학습을 계속하면 네트워크를 구성하는 각 뉴런들은 입력한 데이터에 대해 특정 패턴을 잘 추출할 수 있도록 적응하게 됩니다.

<sup>3</sup> 더 다양한 컨볼루션 필터는 다음 링크에서 찾아볼 수 있습니다. [https://en.wikipedia.org/wiki/Kernel\\_\(image\\_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing))

<sup>4</sup> Kai Yu & Andrew Ng, 《Feature Learning for Image Classification》, ECCV2010, <http://bitly/2Lfzxyx>

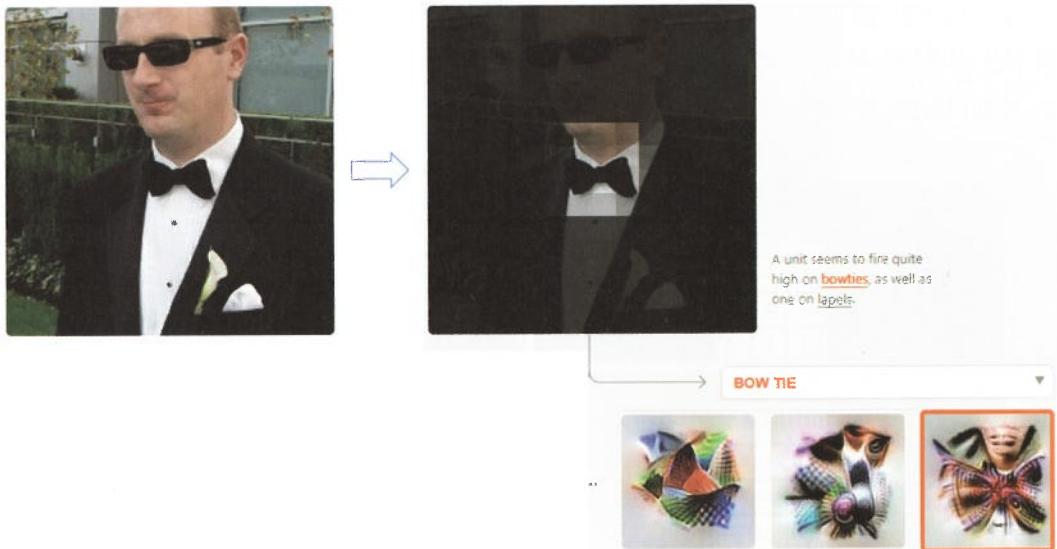


그림 6.4 나비 넥타이에 반응하는 뉴런(우측 하단)은 이미지의 나비 넥타이를 정확히 찾습니다.<sup>5</sup>

컨볼루션 신경망은 어떻게 특징을 자동으로 추출하는 것일까요? 다음 절에서 컨볼루션 신경망의 주요 레이어를 하나씩 살펴보며 그 방법을 알아보겠습니다.

## 6.2 주요 레이어 정리

지금까지 이 책에 나온 레이어는 Dense 레이어와 Flatten 레이어의 두 종류였습니다. Dense 레이어는 신경망에서 가장 기본이 되는 레이어로, 각 뉴런이 서로 완전히 연결되기 때문에 완전 연결(Fully-connected) 레이어라고도 합니다. Flatten 레이어는 다차원의 이미지를 1차원으로 평평하게 바꿔주는 단순한 레이어입니다.

<sup>5</sup> <https://distill.pub/2018/building-blocks/>

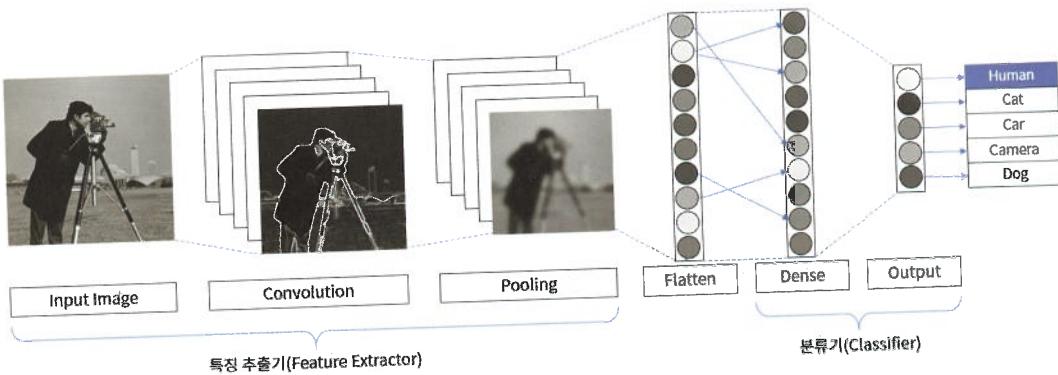


그림 6.5 이미지 분류에 사용되는 컨볼루션 신경망의 구조<sup>6</sup>

그림 6.5에 이미지 분류에 사용되는 일반적인 컨볼루션 신경망의 구조가 나와 있습니다. 분류를 위한 커널은 특징 추출기(Feature Extractor)와 분류기(Classifier)가 합쳐져 있는 형태입니다. 이 컨볼루션 신경망은 특징 추출기(Feature Extractor)와 분류기(Classifier)가 합쳐져 있는 형태입니다. 이 컨볼루션 신경망은 특징 추출기의 역할을 하는 것은 컨볼루션 레이어와 폴링 레이어이며, Dense 레이어는 분류기의 역할을 합니다.

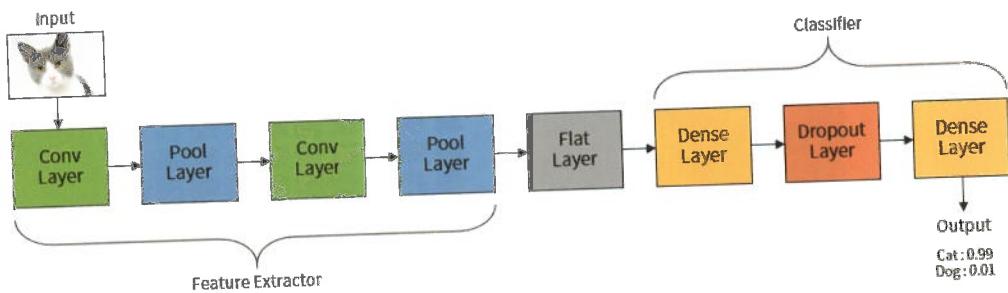


그림 6.6 이미지 분류에 사용되는 컨볼루션 신경망의 구조

그림 6.6에서는 각 레이어의 이름만 나열해서 간결한 그림을 만들었습니다. 특징 추출기에는 컨볼루션 레이어와 폴링 레이어가 교차되며 배치됩니다. 분류기에는 Dense 레이어가 배치되며, 과적합을 막기 위해서 드롭아웃 레이어가 Dense 레이어 사이에 배치됩니다. 마지막 Dense 레이어 뒤에는 드롭아웃 레이어가 배치되지 않습니다.

<sup>6</sup> 출처: Convolutional Neural Networks for Image Classification – General Architecture of a Convolutional Neural Network, compleatge, <http://bit.ly/2LWauQq>

컨볼루션 신경망의 구조에 대한 큰 그림을 그렸으니 이제 새롭게 나온 세 개의 레이어를 하나씩 살펴보겠습니다.

### 6.2.1 컨볼루션 레이어

컨볼루션 레이어(Convolution Layer)는 말 그대로 컨볼루션 연산을 하는 레이어입니다. 다만 여기서 사용하는 필터는 사람이 미리 정해놓는 것이 아니라 네트워크의 학습을 통해 자동으로 추출됩니다. 따라서 코드에서 지정해야 하는 값은 필터를 채우는 각 픽셀의 값이 아니라 필터의 개수 정도입니다. 그림 6.7에서 볼 수 있듯이 필터의 수가 많으면 다양한 특징을 추출할 수 있습니다.

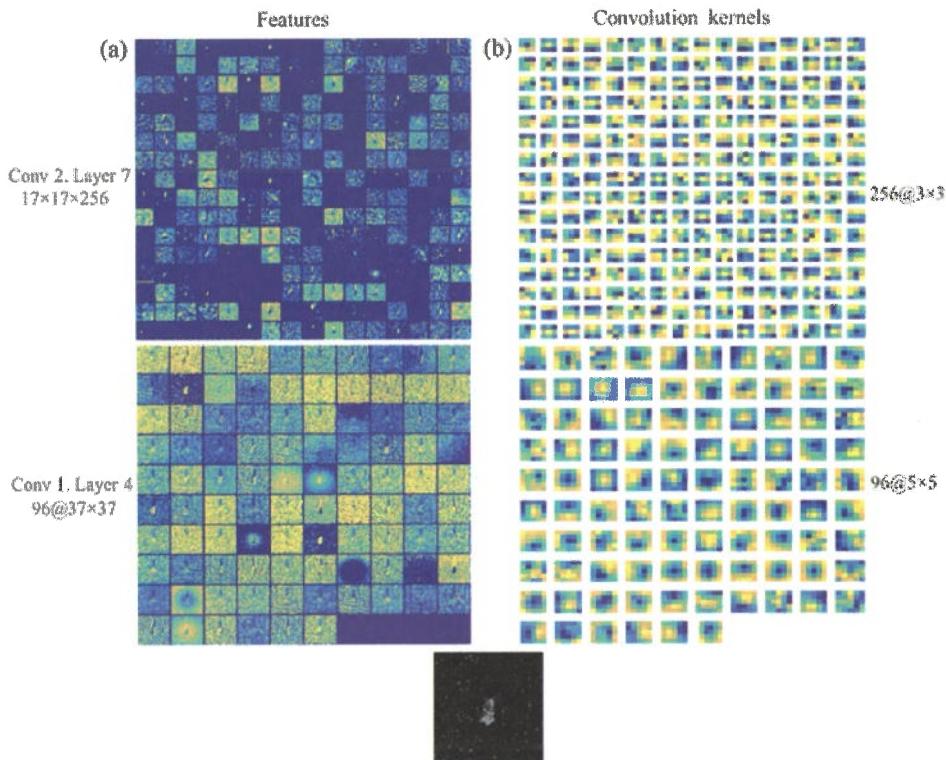


그림 6.7 오른쪽은 필터, 왼쪽은 필터로 추출한 중간 계산 이미지(특징)입니다.<sup>7</sup>

<sup>7</sup> 출처: Moussa Amrani & Feng Jiang, Deep feature extraction and combination for synthetic aperture radar target classification, Journal of Applied Remote Sensing 11(04)1, <http://bit.ly/2XEexrK>

컨볼루션 레이어는 작게는 1차원부터 크게는 3차원 이상까지 다양한 차원으로 사용할 수 있지만 여기서는 가장 기본이 되는 2차원을 기준으로 설명하겠습니다.

이미지에는 원색으로 구성된 채널(Channel)이 있습니다. 채널이란 각 이미지가 가진 색상에 대한 정보를 분리해서 담아놓는 공간입니다. 흑백 이미지는 각 픽셀에 대한 정보가 담긴 채널이 하나뿐이고, 보통의 컬러 이미지는 빨강(Red), 초록(Green), 파랑(Blue)의 삼원색으로 된 세 개의 채널을 가지고 있습니다.

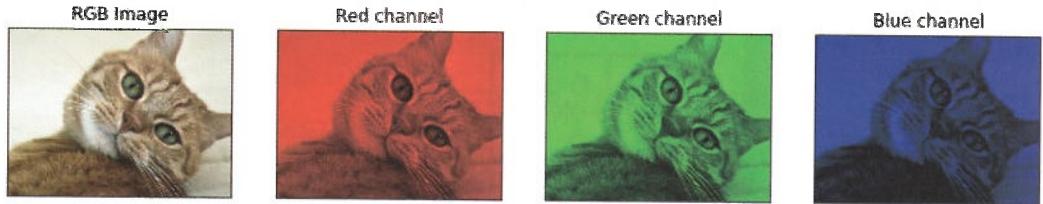


그림 6.8 컬러 이미지에서 분리된 Red, Green, Blue 채널 이미지

컨볼루션 레이어는 각 채널에 대해 계산된 값을 합쳐서 새로운 이미지를 만들어냅니다. 이때 새로운 이미지의 마지막 차원 수는 필터의 수와 동일합니다. 일반적인 컨볼루션 신경망은 여러 개의 컨볼루션 레이어를 쌓으면서 뒤쪽 레이어로 갈수록 필터의 수를 점점 늘리기 때문에 이미지의 마지막 차원 수는 점점 많아집니다.

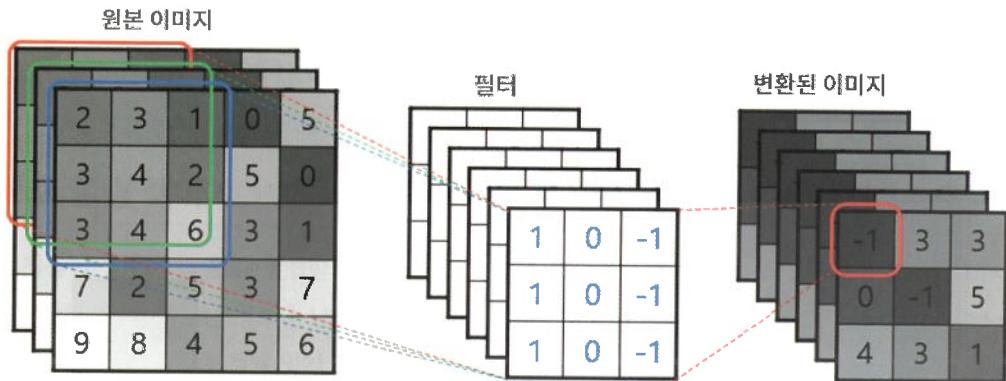


그림 6.9 RGB 채널을 가진 이미지에 컨볼루션 연산을 적용한 결과

컨볼루션 레이어는 다른 레이어와 마찬가지로 `tf.keras.layers`에서 임포트할 수 있습니다. 2차원 이미지를 다루는 컨볼루션 레이어를 생성하는 코드는 다음과 같습니다.

### 예제 6.1 Conv2D 레이어 생성 코드

```
conv1 = tf.keras.layers.Conv2D(kernel_size=(3,3),strides=(2,2),padding='valid',filters=16)
```

Conv2D 레이어를 생성할 때의 주요 인수는 `kernel_size`, `strides`, `padding`, `filters`의 네 가지입니다.

`kernel_size`는 필터 행렬의 크기입니다. 이것은 수용 영역(receptive field)이라고도 부릅니다. 앞의 숫자는 높이, 뒤의 숫자는 너비이고 숫자를 하나만 쓸 경우 높이와 너비를 동일한 값으로 사용한다는 뜻입니다.

`strides`는 필터가 계산 과정에서 한 스텝마다 이동하는 크기입니다. 기본값은 (1,1)이고 (2,2) 등으로 설정할 경우 한 칸씩 전너뛰면서 계산하게 됩니다. `kernel_size`와 마찬가지로 앞의 숫자는 높이, 뒤의 숫자는 너비이고 숫자를 하나만 쓸 경우 높이와 너비는 동일한 값입니다. 그림 6.10처럼 동일한 조건에서 `strides`가 달라지면 결과 이미지의 크기에 영향을 줍니다.

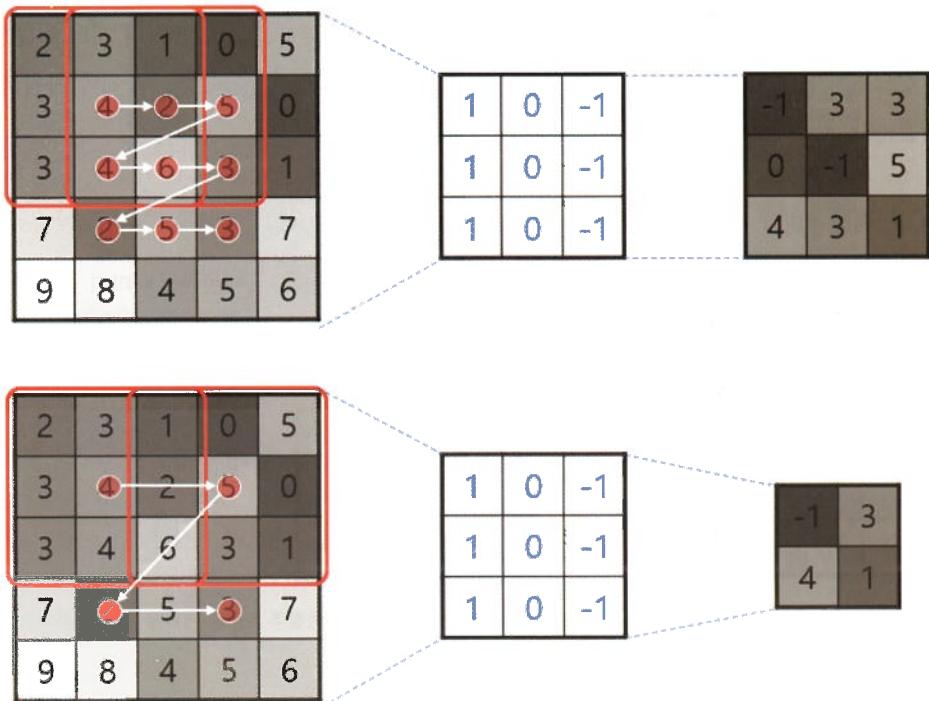


그림 6.10 `strides=1` 때와 `strides=2` 때의 결과 이미지 비교. 붉은색 원은 필터의 중심

padding은 컨볼루션 연산 전에 입력 이미지 주변에 빈 값을 넣을지 지정하는 옵션으로서 'valid'와 'same'이라는 2가지 옵션 중 하나를 사용합니다. 'valid'는 우리가 봄 것과 동일한 방식으로 빈 값을 사용하지 않습니다. 'same'은 빈 값을 넣어서 출력 이미지의 크기를 입력과 같도록 보존합니다. 이때 빈 값으로 0이 쓰이는 경우에 제로 패딩(zero padding)이라고 부릅니다.

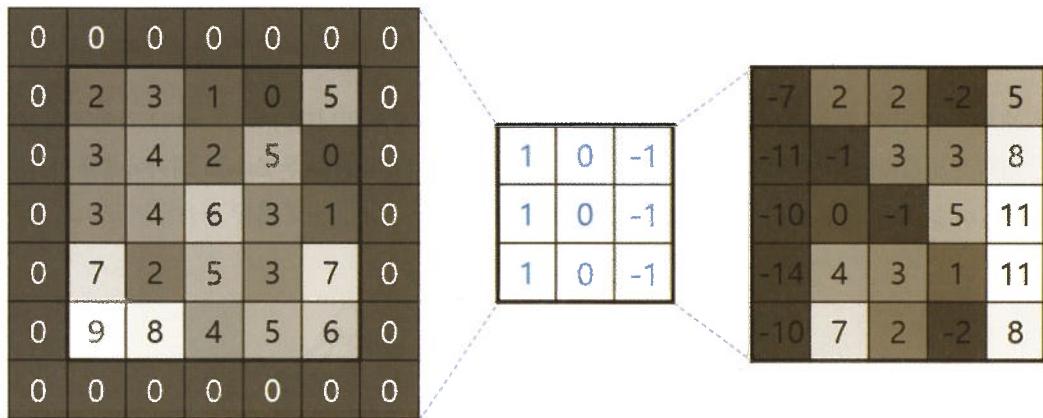


그림 6.11 padding='same'일 때 입력 이미지와 출력 이미지의 크기가 같아집니다.

filters는 필터의 개수입니다. 필터의 개수는 네트워크가 얼마나 많은 특징을 추출할 수 있는지 결정하기 때문에 많을수록 좋지만, 너무 많을 경우 학습 속도가 느려질 수 있고 과적합이 발생할 수도 있습니다. 가장 유명한 컨볼루션 신경망 네트워크 중 하나인 VGG는 네트워크가 깊어질수록 필터의 수를 2배씩 늘려 나갑니다( $64 \rightarrow 128 \rightarrow 256 \rightarrow 512$ ).

## 6.2.2 풀링 레이어

이미지를 구성하는 픽셀 중 인접한 픽셀들은 비슷한 정보를 갖고 있는 경우가 많습니다. 이런 이미지의 크기를 줄이면서 중요한 정보만 남기기 위해 서브샘플링(subsampling)이라는 기법을 사용합니다. 컴퓨터의 메모리 크기는 한정돼 있기 때문에 중요한 정보만 남기는 과정은 효율적인 메모리 사용에 도움이 되고, 또 계산할 정보가 줄어들기 때문에 과적합을 방지하는 효과도 있습니다. 이 과정에 사용되는 레이어가 풀링 레이어(Pooling Layer)입니다.

풀링 레이어에는 Max 풀링 레이어, Average 풀링 레이어 등이 있습니다. 이 가운데 컨볼루션 레이어에는 Max 풀링 레이어가 더 많이 쓰입니다. 예제 6.2는 Conv2D 레이어와 같이 쓰이는 MaxPool2D 레이어의 생성 코드입니다.

#### 예제 6.2 MaxPool2D 레이어 생성 코드

```
pool1 = tf.keras.layers.MaxPool2D(pool_size=(2,2), strides=(2,2))
```

두 개의 주요 인수 중 pool\_size는 한 번에 Max 연산을 수행할 범위입니다. pool\_size=(2,2)이므로 높이 2, 너비 2의 사각형 안에서 최댓값만 남기는 연산을 하게 됩니다. strides는 Conv2D 레이어에서 나온 것과 동일하게 계산 과정에서 한 스텝마다 이동하는 크기입니다.

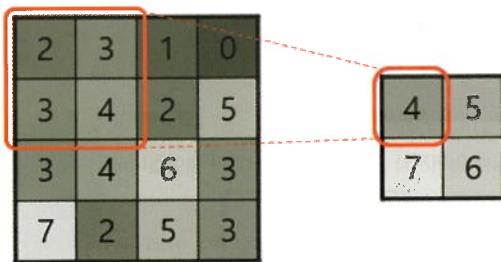


그림 6.12 pool\_size=(2,2), strides=(2,2) 일 때의 MaxPool2D 레이어 적용 결과

pool\_size=(2,2)와 strides=(2,2) 옵션으로 MaxPool2D 레이어를 사용하면 결과 이미지의 크기는 너비, 높이가 각각 절반으로 줄어듭니다. 홀수일 때는 절반에서 내림한 값이 됩니다(예: 5→2).

풀링 레이어에는 가중치가 존재하지 않기 때문에 학습되지 않으며, 네트워크 구조에 따라 생략되기도 합니다.

### 6.2.3 드롭아웃 레이어

드롭아웃 레이어(Dropout Layer)는 네트워크의 과적합을 막기 위한 딥러닝 연구자들의 노력의 결실 중 하나로, 토론토 대학의 제프리 힌튼 교수팀이 발표했습니다.<sup>8</sup> 레딧(reddit)의 머신러닝 커뮤니티에서

<sup>8</sup> G.E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R.R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. <https://arxiv.org/abs/1207.0580>

2016년에 진행됐던 구글 브레인 팀의 AMA(Ask Me Anything) 스레드<sup>9</sup>에서 한 유저가 물어본 드롭아웃 레이어의 발견에 대한 질문에 제프리 힌튼 교수는 다음과 같이 대답했습니다.

나는 은행에 갔다. 창구 직원이 계속 바뀌어서 그 이유를 물어보자, 그들은 이유는 모르겠지만 어쨌든 인사 이동이 많다고 했다. 아마도 직원들 간의 협력이 은행에서 부정을 저지를 수 있기 때문에 그것을 막기 위한 조치라고 생각되었다. 이 사실은 나에게 학습 과정에서 무작위로 뉴런의 부분집합을 제거하는 것이 뉴런들간의 공모(conspiracy)를 막고 과적합(overfitting)을 감소시킬 것이라고 깨닫게 했다.<sup>10</sup>

제프리 힌튼 교수는 평소에 네트워크의 학습에 대해 많이 고민했기 때문에 일상에서 마주치는 가벼운 사장을 보고도 깨달음을 얻은 것이 아닐까,라는 생각이 듭니다.

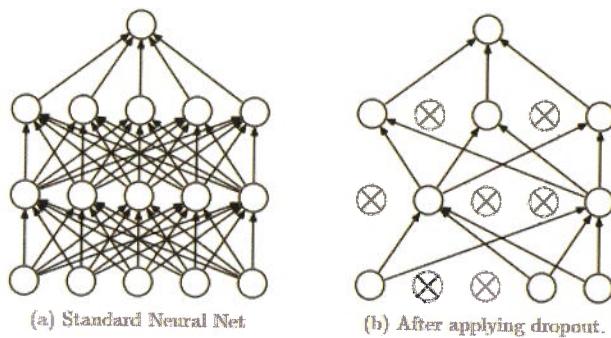


그림 6.13 드롭아웃 레이어의 동작 구조<sup>11</sup>

드롭아웃 레이어는 앞에서 인용한 것처럼 학습 과정에서 무작위로 뉴런의 부분집합을 제거하는 것입니다. 네트워크가 학습할 때 같은 레이어에 있는 뉴런들은 결맞값에 의해 서로 같은 영향을 받습니다. 따라서 각 뉴런의 계산 결과를 모두 더해서 나오는 결맞값은 한쪽으로 치우치게 됩니다. 이를 막기 위한 드롭아웃 레이어는 학습 과정에서는 확률적으로 일부 뉴런에 대한 연결을 끊고, 테스트할 때는 정상적으로 모든 값을 포함해서 계산합니다. 예제 6.3은 드롭아웃 레이어의 생성 코드입니다.

<sup>9</sup> <http://bit.ly/2XExL01>

<sup>10</sup> <http://bit.ly/2LWaIXM>

<sup>11</sup> Srivastava, Nitish, et al, Dropout: a simple way to prevent neural networks from overfitting, JMLR 2014

```
pool1 = tf.keras.layers.Dropout(rate=0.3)
```

주요 인수는 `rate`로, 제외할 뉴런의 비율을 나타냅니다. 간단한 레이어이지만 AlexNet, VGG, GoogLeNet, DenseNet 등 거의 모든 주요 컨볼루션 신경망에 사용됐습니다. 드롭아웃 레이어도 가치가 없기 때문에 학습되지 않습니다.

이렇게 컨볼루션 신경망을 구성하는 레이어 중 가장 중요한 세 가지, 컨볼루션 레이어, 풀링 레이어, 드롭아웃 레이어에 대해 알아봤습니다. 정리하면 컨볼루션 레이어는 특징을 추출하는 역할, 풀링 레이어는 중요한 정보만 남기고 계산 부담을 줄여주는 역할, 드롭아웃 레이어는 과적합을 방지하는 역할을 합니다. 이 가운데 학습이 가능한 것은 컨볼루션 레이어뿐입니다.

이제 구글 코랩에서 실제 코드와 함께 컨볼루션 신경망을 직접 학습시켜 보겠습니다.

## 6.3 Fashion MNIST 데이터세트에 적용하기

이번 장은 앞에서 두 절에 걸쳐 이론만 설명했기 때문에 실전에 즉시 활용할 수 있는 코드 예제를 기다리셨던 분들에게는 약간 지루했을지도 모르겠습니다. 하지만 컨볼루션 신경망의 기본 원리와 자주 쓰이는 레이어에 대해 알아두는 것은 매우 중요하기 때문에 앞의 두 절을 할애해서 이론을 설명한 것입니다.

그럼 이제 실전 코드를 살펴보겠습니다. 5장의 후반부에 나온 Fashion MNIST 데이터를 다시 사용합니다. 5장에서는 주로 Dense 레이어를 사용해서 Fashion MNIST의 분류 문제를 풀었다면, 6장에서는 새롭게 배운 컨볼루션 레이어와 풀링 레이어 등을 사용해서 분류 문제를 풀 때 퍼포먼스가 얼마나 개선될 수 있는지 알아보겠습니다.

```
import tensorflow as tf

fashion_mnist = tf.keras.datasets.fashion_mnist
(train_X, train_Y), (test_X, test_Y) = fashion_mnist.load_data()

train_X = train_X / 255.0
test_X = test_X / 255.0
```

tf.keras에서 Fashion MNIST 데이터세트를 불러오고 정규화하는 부분은 5장과 동일합니다. 그런데 Dense 레이어에 훈련 데이터와 테스트 데이터를 통과시켰던 5장과 달리 6장에서는 Conv2D 레이어로 컨볼루션 연산을 해야 합니다. 이미지는 보통 채널을 가지고 있고(컬러 이미지는 RGB의 3채널, 흑백 이미지는 1채널), Conv2D 레이어는 채널을 가진 형태의 데이터를 받도록 기본적으로 설정돼 있기 때문에 예제 6.5에서는 채널을 갖도록 데이터의 Shape을 바꿉니다.

#### 예제 6.5 Fashion MNIST 데이터세트 불러오기 및 정규화

##### [IN]

```
# reshape 이전
print(train_X.shape, test_X.shape)

train_X = train_X.reshape(-1, 28, 28, 1)
test_X = test_X.reshape(-1, 28, 28, 1)

# reshape 이후
print(train_X.shape, test_X.shape)
```

##### [OUT]

```
(60000, 28, 28) (10000, 28, 28)
(60000, 28, 28, 1) (10000, 28, 28, 1)
```

Fashion MNIST 데이터를 구성하는 흑백 이미지는 1개의 채널을 갖기 때문에 `reshape()` 함수를 사용해 데이터의 가장 뒤쪽에 채널 차원을 추가합니다. 이 작업으로 데이터 수는 달라지지 않습니다.  $60000 \times 28 \times 28$ 과  $60000 \times 28 \times 28 \times 1$ 이 같은 값인 것처럼 말입니다.

데이터를 확인하기 위해 `matplotlib.pyplot`을 사용해 그래프를 그려볼 수 있습니다.

#### 예제 6.6 데이터 확인

##### [IN]

```
import matplotlib.pyplot as plt
# 전체 그래프의 크기를 width=10, height=10으로 지정합니다.
plt.figure(figsize=(10, 10))
for c in range(16):
    # 4행 4열로 지정한 그리드에서 c+1번째의 칸에 그래프를 그립니다. 1~16번째 칸을 채우게 됩니다.
    plt.subplot(4,4,c+1)
```

```

plt.imshow(train_X[c].reshape(28,28), cmap='gray')

plt.show()

# 훈련 데이터의 첫 번째 ~ 16번째 까지의 라벨을 프린트합니다.
print(train_Y[:16])

```

**[OUT]**



그래프를 그리기 위한 데이터는 2차원이어야 하기 때문에 `plt.imshow(train_X[c].reshape(28,28), cmap='gray')`에서 각 데이터를 대상으로 `reshape()` 함수를 츠해 3차원 데이터를 다시 2차원 데이터로 변환합니다. 각 이미지는 `plt.subplot()` 함수에서 지정되는 그리드(grid)의 각 칸에 위에서 아래, 왼쪽에서 오른쪽 순서대로 그려집니다. 데이터의 범주 중 9번은 신발, 0번은 티셔츠/상의, 3번은 드레스입니다. 출력 이미지 첫 번째 행의 4개 이미지가 9, 0, 0, 3의 라벨로 잘 분류돼 있는 것을 확인할 수 있습니다.

표 6.1 Fashion MNIST의 범주

리벨	범주
0	티셔츠/상의
1	바지
2	스웨터
3	드레스
4	코트
5	샌들
6	셔츠
7	운동화
8	기방
9	부츠

이제 모델을 생성할 차례입니다. 먼저 비교를 위해 풀링 레이어 없이 컨볼루션 레이어만 사용한 모델을 정의해보겠습니다.

예제 6.7 Fashion MNIST 분류를 위한 컨볼루션 신경망 모델 정의

[IN]

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(input_shape=(28,28,1), kernel_size=(3,3), filters=16),
    tf.keras.layers.Conv2D(kernel_size=(3,3), filters=32),
    tf.keras.layers.Conv2D(kernel_size=(3,3), filters=64),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=128, activation='relu'),
    tf.keras.layers.Dense(units=10, activation='softmax')
])

model.compile(optimizer=tf.keras.optimizers.Adam(),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.summary()
```

**[OUT]**

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 16)	160
conv2d_1 (Conv2D)	(None, 24, 24, 32)	4640
conv2d_2 (Conv2D)	(None, 22, 22, 64)	18496
flatten (Flatten)	(None, 30976)	0
dense (Dense)	(None, 128)	3965056
dense_1 (Dense)	(None, 10)	1290

Total params: 3,989,642

Trainable params: 3,989,642

Non-trainable params: 0

모델에는 총 3개의 Conv2D 레이어를 사용했고 그중 첫 레이어의 input\_shape은 (28,28,1)로 입력 이미지의 높이, 너비, 채널 수를 정의하고 있습니다. 필터의 수는 16, 32, 64로 뒤로 갈수록 2배씩 늘렸습니다. Flatten 레이어로 다차원 데이터를 1차원으로 정렬한 다음에 2개의 Dense 레이어를 사용해 분류기를 만들었습니다. 그럼 이 모델의 퍼포먼스를 확인해보겠습니다.

그런데 그 전에 먼저 해야 할 작업이 있습니다. 이 책에서는 뒤로 갈수록 점점 복잡하고 파라미터 개수가 많은 네트워크를 다루기 때문에 구글 코랩에서 하드웨어 가속기를 쓰지 않으면 계산이 몹시 느릴 수 있습니다. 하드웨어 가속기를 사용하려면 메뉴에서 ‘런타임’ → ‘런타임 유형 변경’ → ‘하드웨어 가속기’를 차례로 선택한 후 ‘GPU’를 지정하면 됩니다. TPU를 쓸 수도 있지만 TPU를 쓰기 위해 현재 2.0 버전에서는 코드 수정이 필요하고, 병렬 처리를 필요로 하는 큰 데이터를 사용할 때 TPU의 병렬 처리가 빛을 빛하는데, 이 책에는 그 정도의 큰 예제는 다루지 않으므로 여기서는 GPU를 사용하겠습니다. GPU로 설정한 뒤에는 우측 하단의 ‘저장’ 버튼을 눌러서 해당 설정을 적용합니다.

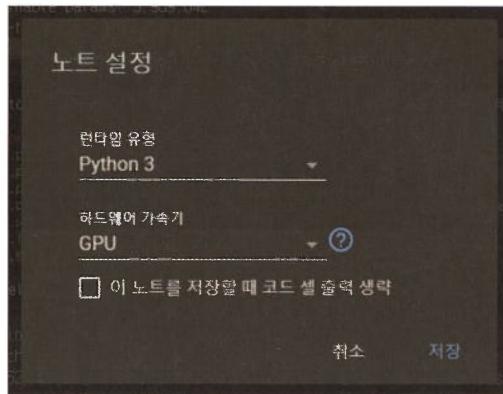


그림 6.14 구글 코랩에서 코드를 실행 전에 하드웨어 가속기를 GPU로 설정합니다.

그런데 여기서 사용되는 GPU는 어떤 것일까요? 간단한 배시 셸 명령어로 GPU의 사양을 확인할 수 있습니다.

#### 예제 6.8 구글 코랩의 GPU 사양 확인

##### [IN]

```
!nvidia-smi
```

##### [OUT]

```
Sun Jun 30 14:00:21 2019

NVIDIA-SMI 418.67 Driver Version: 410.79 CUDA Version: 10.0
+-----+
| GPU Name Persistence-M| Bus-Id Disp.A | Volatile Uncorr. ECC |
| Fan Temp Perf Pwr:Usage/Cap| Memory-Usage | GPU-Util Compute M |
| +--+ +--+ +--+ +--+ +--+ +--+ +--+ +--+ +--+ +--+ +--+ +--+
| 0 Tesla K80 Off 00000000:00:04.0 Off 0MiB / 11441MiB | 0% Default |
| N/A 29C P8 27W / 149W | +-----+
+-----+

Processes:
+----+----+----+----+----+----+
| GPU PID Type Process name | GPU Memory |
| +--+ +--+ +--+ +--+ +--+ +--+ |
| No running processes found | Usage       |
+----+----+----+----+----+----+
```

Tesla K80을 쓰고 있음을 확인할 수 있습니다. 약 300만원 정도의 가격으로 비싼 가격만큼 훌륭한 퍼포먼스를 보이는 GPU입니다. Tesla K80보다 4배 빠른 Tesla T4가 사용 가능할 때도 있습니다.<sup>12</sup>

Fri Nov 22 05:21:59 2019

```
+-----+
| NVIDIA-SMI 430.50      Driver Version: 418.67      CUDA Version: 10.1      |
+-----+
| GPU  Name      Persistence-M| Bus-Id      Disp.A  | Volatile Uncorr. ECC  |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M.  |
|-----+-----+-----+-----+-----+-----+-----+
|  0  Tesla T4           Off  | 00000000:00:04.0 Off |                  0 |
| N/A   43C    P8    10W /  70W |      0MiB / 15079MiB |     0%      Default |
+-----+-----+-----+-----+-----+-----+-----+
+-----+
| Processes:                               GPU Memory  |
| GPU  PID  Type  Process name          Usage  |
|-----+-----+-----+-----|
| No running processes found            |
+-----+
```

그럼 이제 실제로 모델을 학습시켜 보겠습니다.

#### 예제 6.9 Fashion MNIST 분류를 위한 컨볼루션 신경망 모델 학습

[IN]

```
history = model.fit(train_X, train_Y, epochs=25, validation_split=0.25)

import matplotlib.pyplot as plt
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], 'b-', label='loss')
plt.plot(history.history['val_loss'], 'r--', label='val_loss')
plt.xlabel('Epoch')
plt.legend()
```

<sup>12</sup> 무료 버전의 구글 코랩에서는 GPU를 일정 시간 동안 사용하면 한동안 사용 불가능한 상태가 됩니다. 이때는 CPU로 작업해야 합니다[출처: 자주 묻는 질문, 구글 Colaboratory, <http://bit.ly/2O90Vhl>].

```

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], 'g-', label='accuracy')
plt.plot(history.history['val_accuracy'], 'k--', label='val_accuracy')
plt.xlabel('Epoch')
plt.ylim(0.7, 1)
plt.legend()

plt.show()

model.evaluate(test_X, test_Y, verbose=0)

```

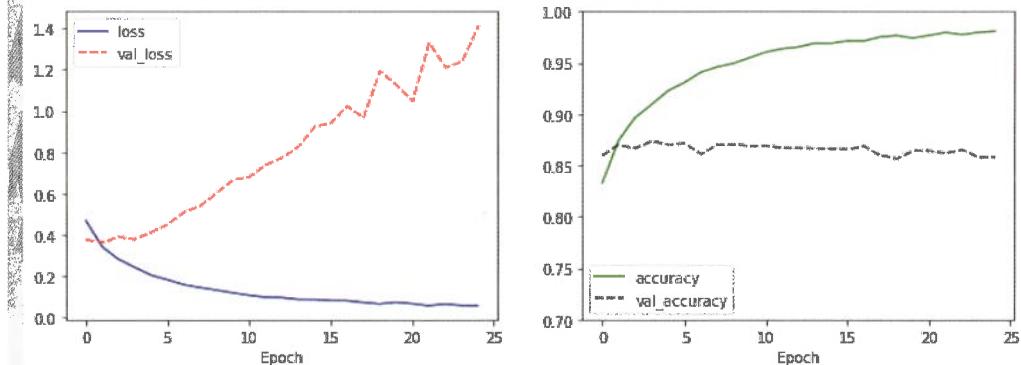
**[OUT]**

```

Train on 45000 samples, validate on 15000 samples
Epoch 1/25
45000/45000 [=====] - 11s 248us/sample - loss: 0.4686 - accuracy: 0.8330 - val_loss: 0.3823 - val_accuracy: 0.8604
Epoch 2/25
45000/45000 [=====] - 7s 152us/sample - loss: 0.3427 - accuracy: 0.8748 - val_loss: 0.3645 - val_accuracy: 0.8703
Epoch 3/25
45000/45000 [=====] - 7s 152us/sample - loss: 0.2828 - accuracy: 0.8965 - val_loss: 0.3946 - val_accuracy: 0.8674
Epoch 4/25
45000/45000 [=====] - 7s 151us/sample - loss: 0.2448 - accuracy: 0.9094 - val_loss: 0.3819 - val_accuracy: 0.8742
Epoch 5/25
45000/45000 [=====] - 7s 150us/sample - loss: 0.2064 - accuracy: 0.9227 - val_loss: 0.4159 - val_accuracy: 0.8711

```

(이하 생략)



[1.477660627841949, 0.8521]

모델의 학습에는 에포크당 7초 정도가 걸립니다. 참고로 같은 환경에서 하드웨어 가속기 없이 실행했을 때는 약 3분~3분 20초 정도가 걸렸습니다. 약 25배~28배 정도의 속도 차이가 나기 때문에 하드웨어 가속기 설정은 필수입니다.

왼쪽 그래프를 확인해보면 loss는 감소하고 val\_loss는 증가하는 전형적인 과적합의 형태를 나타냅니다. 오른쪽 그래프에서는 훈련 데이터에 대한 모델의 정확도인 accuracy가 빠르게 증가하는 데에 비해서 검증 데이터에 대한 정확도인 val\_accuracy는 학습이 진행될수록 오히려 감소합니다.

마지막에 `model.evaluate()` 함수로 계산되는 결과 중 첫 번째가 테스트 데이터의 loss이고 두 번째가 테스트 데이터의 accuracy입니다. 테스트 데이터의 accuracy는 85.21%가 나왔는데, 이는 5.3절의 Dense 레이어 네트워크가 달성한 88.5%에도 못 미치는 수준입니다.

이를 어떻게 개선할 수 있을까요? 일단 6.2절에 나왔던 풀링 레이어와 드롭아웃 레이어를 모두 사용해보겠습니다.

#### 예제 6.10 Fashion MNIST 분류를 위한 컨볼루션 신경망 모델 정의 – 풀링 레이어, 드롭아웃 레이어 추가

##### [IN]

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(input_shape=(28,28,1), kernel_size=(3,3), filters=32),
    tf.keras.layers.MaxPool2D(strides=(2,2)),
    tf.keras.layers.Conv2D(kernel_size=(3,3), filters=64),
    tf.keras.layers.MaxPool2D(strides=(2,2)),
    tf.keras.layers.Conv2D(kernel_size=(3,3), filters=128),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=128, activation='relu'),
    tf.keras.layers.Dropout(rate=0.3),
    tf.keras.layers.Dense(units=10, activation='softmax')
])

model.compile(optimizer=tf.keras.optimizers.Adam(),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.summary()
```

## [OUT]

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_5 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_2 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_6 (Conv2D)	(None, 3, 3, 128)	73856
flatten_1 (Flatten)	(None, 1152)	0
dense_2 (Dense)	(None, 128)	147584
dropout_1 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 10)	1290
Total params: 241,546		
Trainable params: 241,546		
Non-trainable params: 0		

summary() 함수로 레이어 구조를 확인해보면 총 파라미터 개수가 예제 6.7의 400만에 가까운 숫자에 비해 24만 정도로, 약 6%로 줄어들었습니다. 이는 풀링 레이어가 이미지의 크기를 줄여주고 있기 때문에 Flatten 레이어에 들어온 파라미터 수가 1,152로 예제 6.7의 30,976에 비해 감소했기 때문입니다. 두 모델에서 가장 많은 파라미터가 있는 레이어는 Flatten 레이어 다음의 첫 번째 Dense 레이어이기 때문에 이 레이어에 넘어오는 파라미터 수가 적을수록 전체 파라미터 수도 적어지는 것입니다.

Dense 레이어 사이에는 드롭아웃 레이어도 사용됐습니다. 풀링 레이어와 드롭아웃 레이어는 모두 과적합을 줄이는 데 기여하게 됩니다. 실제로 그렇게 되는지 네트워크 학습을 통해 확인해보겠습니다.

## [IN]

```

history = model.fit(train_X, train_Y, epochs=25, validation_split=0.25)

import matplotlib.pyplot as plt
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], 'b-', label='loss')
plt.plot(history.history['val_loss'], 'r--', label='val_loss')
plt.xlabel('Epoch')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], 'g-', label='accuracy')
plt.plot(history.history['val_accuracy'], 'k--', label='val_accuracy')
plt.xlabel('Epoch')
plt.ylim(0.7, 1)
plt.legend()

plt.show()

model.evaluate(test_X, test_Y, verbose=0)

```

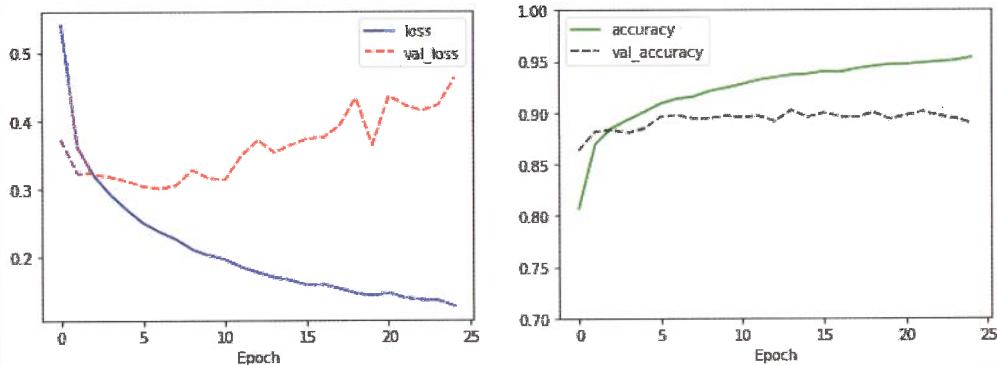
## [OUT]

```

Train on 45000 samples, validate on 15000 samples
Epoch 1/25
45000/45000 [=====] - 6s 135us/sample - loss: 0.5381 - accuracy: 0.8065 - val_loss: 0.3727 - val_accuracy: 0.8635
Epoch 2/25
45000/45000 [=====] - 6s 125us/sample - loss: 0.3622 - accuracy: 0.8697 - val_loss: 0.3239 - val_accuracy: 0.8816
Epoch 3/25
45000/45000 [=====] - 6s 124us/sample - loss: 0.3191 - accuracy: 0.8848 - val_loss: 0.3219 - val_accuracy: 0.8832
Epoch 4/25
45000/45000 [=====] - 6s 126us/sample - loss: 0.2917 - accuracy: 0.8931 - val_loss: 0.3181 - val_accuracy: 0.8806
Epoch 5/25
45000/45000 [=====] - 6s 127us/sample - loss: 0.2696 - accuracy:

```

0.9008 - val\_loss: 0.3124 - val\_accuracy: 0.8846  
(이하 생략)



[0.4727696822822094, 0.8909]

val\_loss가 여전히 증가하고 있지만 val\_accuracy는 일정한 수준에 머무르고 있습니다. 테스트 데이터에 대한 분류 성적은 89.09%가 나옵니다. 폴링 레이어와 드롭아웃 레이어를 쓰지 않았을 때보다 개선된 수치이고, 5장의 Dense 레이어만 사용한 네트워크의 성적도 1% 정도 앞질렀습니다.

하지만 이보다 더 좋은 성과를 낼 수 있을 것 같습니다. Fashion MNIST의 공식 깃허브 저장소에는 테스트 데이터 분류 성적에서 95% 이상을 달성한 몇몇 방법들이 기록돼 있습니다.<sup>13</sup> 이 방법들을 참고해서 현재의 분류 성적을 90% 이상으로 끌어올려보겠습니다.

## 6.4 퍼포먼스 높이기

컨볼루션 신경망에서 퍼포먼스를 높이는 데는 여러 가지 방법이 있지만 그중 대표적이면서 쉬운 두 가지 방법은 ‘더 많은 레이어 쌓기’와 ‘이미지 보강(Image Augmentation)’ 기법입니다.

<sup>13</sup> <http://bit.ly/2YCgQsV>

## 6.4.1 더 많은 레이어 쌓기

컨볼루션 신경망의 역사, 더 나아가 딥러닝의 역사는 더 깊은 신경망을 쌓기 위한 노력이라고 해도 과언이 아닙니다. 딥러닝에서 네트워크 구조를 깊게 쌓는 것이 가능해진 후 딥러닝의 발전을 이끈 컨볼루션 신경망에서는 컨볼루션 레이어가 중첩된 더 깊은 구조가 계속해서 나타났고, 그럴 때마다 이전 구조의 퍼포먼스를 크게 개선했습니다.

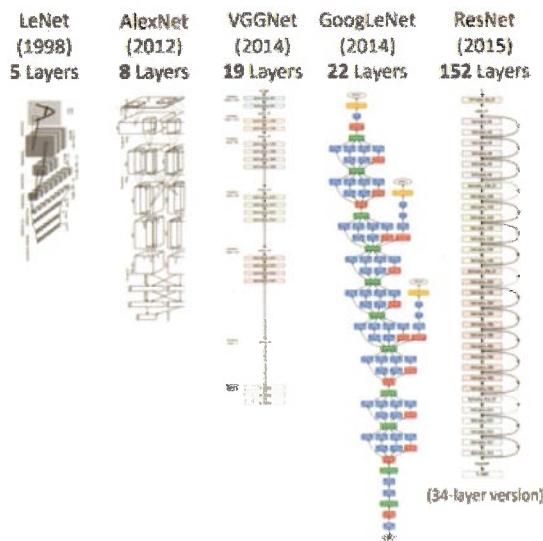


Figure 1. Progression towards deeper neural network structures in recent years (see, e.g., [6], [7], [8], [9], [10]).

그림 6.15 점점 깊어진 컨볼루션 신경망<sup>14</sup>

예제 6.4.1에서는 VGGNet의 스타일로 구성한 컨볼루션 신경망을 사용해 Fashion MNIST 데이터를 분류하는 모델을 정의했습니다. VGG는 단순한 구조이면서도 성능이 괜찮기 때문에 지금도 이미지의 특징 추출을 위한 네트워크에서 많이 쓰이고 있습니다. 유명한 Style Transfer 논문<sup>15</sup>에서도 VGGNet(VGG-19)을 사용했습니다.

<sup>14</sup> 출처: Surat Teerapittayanon et al, Distributed Deep Neural Networks over the Cloud, the Edge and End Devices, ICDCS, 2017. <https://arxiv.org/abs/1709.01921>

<sup>15</sup> Gatys, Leon A., Ecker, Alexander S., and Bethge, Matthias, A neural algorithm of artistic style. <https://arxiv.org/abs/1508.06576>

## [IN]

```

model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(input_shape=(28,28,1), kernel_size=(3,3), filters=32,
    padding='same', activation='relu'),
    tf.keras.layers.Conv2D(kernel_size=(3,3), filters=64, padding='same', activation='relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2,2)),
    tf.keras.layers.Dropout(rate=0.5),
    tf.keras.layers.Conv2D(kernel_size=(3,3), filters=128, padding='same', activation='relu'),
    tf.keras.layers.Conv2D(kernel_size=(3,3), filters=256, padding='valid', activation='relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2,2)),
    tf.keras.layers.Dropout(rate=0.5),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=512, activation='relu'),
    tf.keras.layers.Dropout(rate=0.5),
    tf.keras.layers.Dense(units=256, activation='relu'),
    tf.keras.layers.Dropout(rate=0.5),
    tf.keras.layers.Dense(units=10, activation='softmax')
])

model.compile(optimizer=tf.keras.optimizers.Adam(),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.summary()

```

## [OUT]

Layer (type)	Output Shape	Param #
conv2d_15 (Conv2D)	(None, 28, 28, 32)	320
conv2d_16 (Conv2D)	(None, 28, 28, 64)	18496
max_pooling2d_17 (MaxPooling2D)	(None, 14, 14, 64)	0
dropout_10 (Dropout)	(None, 14, 14, 64)	0
conv2d_17 (Conv2D)	(None, 14, 14, 128)	73856

conv2d_18 (Conv2D)	(None, 12, 12, 256)	295168
max_pooling2d_8 (MaxPooling2D)	(None, 6, 6, 256)	0
dropout_11 (Dropout)	(None, 6, 6, 256)	0
flatten_4 (Flatten)	(None, 9216)	0
dense_10 (Dense)	(None, 512)	4719104
dropout_12 (Dropout)	(None, 512)	0
dense_11 (Dense)	(None, 256)	131328
dropout_13 (Dropout)	(None, 256)	0
dense_12 (Dense)	(None, 10)	2570

Total params: 5,240,842

Trainable params: 5,240,842

Non-trainable params: 0

VGGNet은 여러 개의 구조로 실험했는데 그중 19개의 레이어가 겹쳐진 VGG-19가 제일 깊은 구조입니다. VGG-19는 특징 추출기의 초반에 컨볼루션 레이어를 2개 겹친 뒤 풀링 레이어 1개를 사용하는 패턴을 2차례, 그 후 컨볼루션 레이어를 4개 겹친 뒤 풀링 레이어 1개를 사용하는 패턴을 3차례 반복합니다.

여기서는 대상 이미지가 작기도 하고 연산 능력의 한계도 있어서 컨볼루션 레이어를 2개 겹치고 풀링 레이어를 1개 사용하는 패턴을 2차례 반복했습니다. 그리고 풀링 레이어의 다음에 드롭아웃 레이어를 위치시켜서 과적합을 방지했고, Flatten 레이어 다음에 이어지는 3개의 Dense 레이어 사이에도 드롭아웃 레이어를 배치했습니다. VGGNet처럼 컨볼루션 레이어와 Dense 레이어의 개수만 세면 VGG-7 정도가 되겠습니다.

오리지널 VGG-19보다는 깊이가 얕지만 총 파라미터 개수는 520만 개로 적지 않습니다. 예제 6.10의 24만 개보다는 약 20배 이상 증가한 숫자입니다. 그럼 이 모델의 성능은 어떤지 학습으로 알아보겠습니다.

## [IN]

```

history = model.fit(train_X, train_Y, epochs=25, validation_split=0.25)

import matplotlib.pyplot as plt
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], 'b-', label='loss')
plt.plot(history.history['val_loss'], 'r--', label='val_loss')
plt.xlabel('Epoch')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], 'g-', label='accuracy')
plt.plot(history.history['val_accuracy'], 'k--', label='val_accuracy')
plt.xlabel('Epoch')
plt.ylim(0.7, 1)
plt.legend()

plt.show()

model.evaluate(test_X, test_Y, verbose=0)

```

## [OUT]

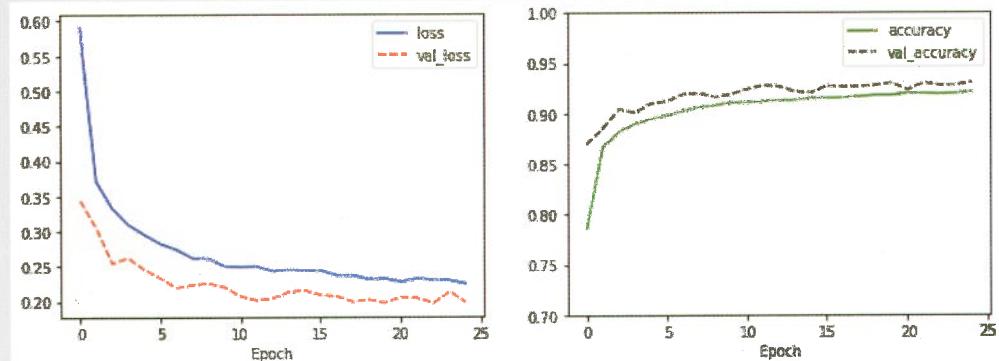
```

Train on 45000 samples, validate on 15000 samples
Epoch 1/25
45000/45000 [=====] - 10s 230us/sample - loss: 0.5885 - accuracy: 0.7862 - val_loss: 0.3433 - val_accuracy: 0.8699
Epoch 2/25
45000/45000 [=====] - 9s 210us/sample - loss: 0.3705 - accuracy: 0.8666 - val_loss: 0.3038 - val_accuracy: 0.8851
Epoch 3/25
45000/45000 [=====] - 9s 209us/sample - loss: 0.3324 - accuracy: 0.8814 - val_loss: 0.2540 - val_accuracy: 0.9039
Epoch 4/25
45000/45000 [=====] - 9s 210us/sample - loss: 0.3091 - accuracy: 0.8892 - val_loss: 0.2616 - val_accuracy: 0.9007
Epoch 5/25
45000/45000 [=====] - 9s 208us/sample - loss: 0.2946 - accuracy:

```

0.8941 - val\_loss: 0.2448 - val\_accuracy: 0.9097

(이하 생략)



[0.21326058280467988, 0.9252]

드디어 val\_loss가 잘 증가하지 않는 훌륭한 그래프를 얻었습니다. 테스트 데이터에 대한 분류 성적도 92.52%로 지금까지 거둔 성적 가운데 제일 우수합니다. 모델은 아직 과적합되지 않았기 때문에 에포크 수를 늘려서 좀 더 돌려볼 수도 있을 것 같습니다.

이로써 간단한 네트워크 구조 변경만으로도 Fashion MNIST 데이터의 분류 성능을 어느 정도 올릴 수 있다는 점을 확인했습니다.

#### 6.4.2 이미지 보강

이미지 보강(Image Augmentation)은 훈련 데이터에 없는 이미지를 새롭게 만들어내서 훈련 데이터를 보강하는 것입니다. 이때 새로운 이미지는 훈련 데이터의 이미지를 원본으로 삼고 일정한 변형을 가해서 만들어집니다.

예를 들어, 다음과 같은 신발 이미지가 훈련 데이터에 있을 때 신발코가 왼쪽을 향하는 이미지만 훈련 데이터에 있고 테스트 데이터에는 신발코가 오른쪽을 향하는 이미지가 있을 경우, 컨볼루션 신경망은 테스트 데이터에서 새롭게 나오는 이미지에 대해 좋은 퍼포먼스를 내지 못합니다. 이때 이미지를 가로로 뒤집어서(horizontal flip) 신발코가 오른쪽을 향하는 이미지도 만들고, 약간 회전시키거나(rotate), 기울이거나(shear), 일부 확대하거나(zoom), 평행이동시켜서(shift) 다양한 이미지를 만들어내서 훈련 데이터의 표현력을 더 좋게 만드는 것입니다.

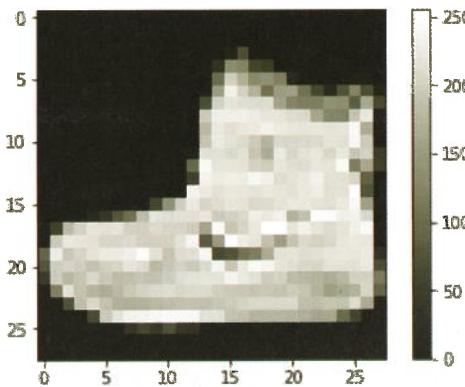


그림 6.16 Fashion MNIST 훈련 데이터의 첫 번째 이미지

tf.keras에는 이러한 이미지 보강 작업을 쉽게 해주는 `ImageDataGenerator`가 있습니다. 예제 6.14에서는 이를 활용해 훈련 데이터의 첫 번째 이미지를 변형시킵니다.

#### 예제 6.14 Image Augmentation 데이터 표시

##### [IN]

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import numpy as np

image_generator = ImageDataGenerator(
    rotation_range=10,
    zoom_range=0.10,
    shear_range=0.5,
    width_shift_range=0.10,
    height_shift_range=0.10,
    horizontal_flip=True,
    vertical_flip=False)

augment_size = 100

x_augmented =
    image_generator.flow(np.tile16(train_X[0].reshape(28*28), 100).reshape(-1, 28, 28, 1),
    np.zeros(augment_size), batch_size=augment_size, shuffle=False).next()[0]
```

<sup>16</sup> `numpy.tile(A, reps)`은 A를 reps에 정해진 형식만큼 반복한 값을 반환합니다. 여기서는 reps가 100이기 때문에 A를 100번 반복한 값을 반환하게 됩니다.

```
# 새롭게 생성된 이미지 표시
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 10))
for c in range(100):
    plt.subplot(10,10,c+1)
    plt.axis('off')
    plt.imshow(x_augmented[c].reshape(28,28), cmap='gray')
plt.show()
```

[OUT]

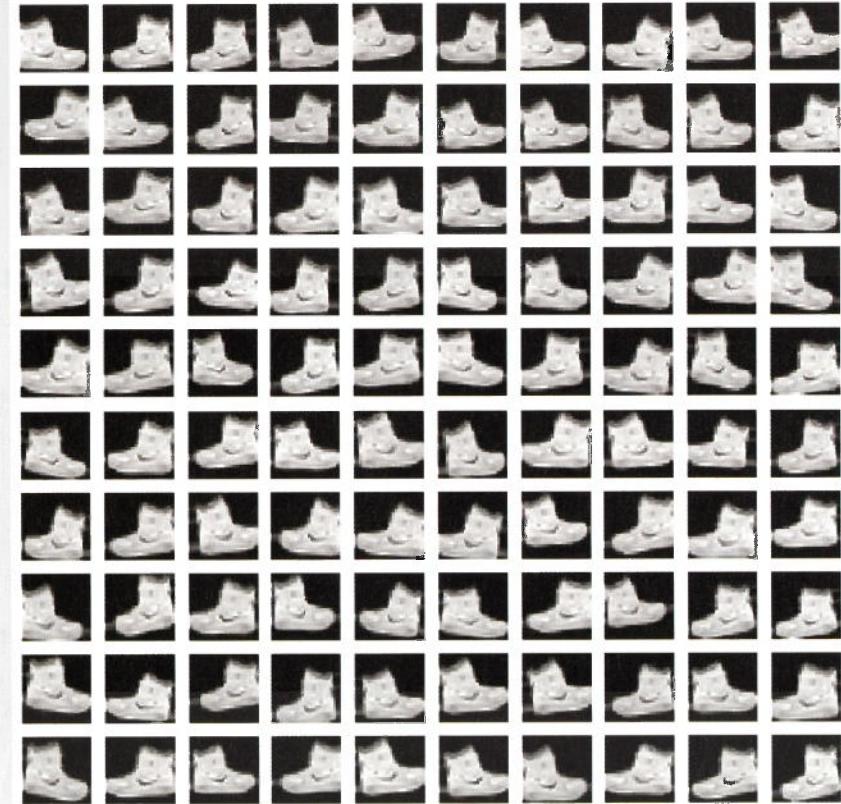


그림 6.16의 신발 이미지가 여러 가지 형태로 조금씩 바뀐 것을 확인할 수 있습니다. 다양한 변형이 가능하기 때문에 각 이미지는 비슷하면서도 서로 조금씩 다릅니다.

`ImageDataGenerator`의 주요 인수들은 `rotation_range`, `zoom_range`, `shear_range` 등입니다. 가로축으로 이미지를 뒤집는 `horizontal_flip`은 사용하지만 세로축으로 뒤집는 `vertical_flip`은 사용하지 않습니다. Fashion MNIST에는 보통 이미지가 위아래로 반듯하게 놓여 있기 때문에 `vertical_flip` 옵션을 `True`로 설정하면 대비하지 않아도 될 경우(이미지가 뒤집혀 있는 경우)에 대해서도 대비하게 되어 퍼포먼스가 뛰어집니다.

`flow()` 함수는 실제로 보강된 이미지를 생성합니다. 이 함수는 `Iterator`라는 객체를 만드는데, 이 객체에 시는 값을 순차적으로 꺼낼 수 있습니다. 값을 꺼내는 방법은 `next()` 함수를 사용하는 것입니다.<sup>17</sup> 한번에 생성할 이미지의 양인 `batch_size`를 위에서 설정한 `augment_size`와 같은 100으로 설정했기 때문에 `next()` 함수로 꺼내는 이미지는 100장이 됩니다. 나머지 부분은 `matplotlib.pyplot`으로 생성된 보강 이미지를 그래프로 그려주는 부분입니다.

그럼 실제로 훈련 데이터 이미지를 보강하기 위해 다양한 이미지를 생성하고 학습을 위해 훈련 데이터에 추가해 보겠습니다.

#### 예제 6.15 이미지 보강

##### [IN]

```
image_generator = ImageDataGenerator(  
    rotation_range=10,  
    zoom_range=0.10,  
    shear_range=0.5,  
    width_shift_range=0.10,  
    height_shift_range=0.10,  
    horizontal_flip=True,  
    vertical_flip=False)  
  
augment_size = 30000  
  
randidx = np.random.randint(train_X.shape[0], size=augment_size)  
x_augmented = train_X[randidx].copy()  
y_augmented = train_Y[randidx].copy()  
x_augmented = image_generator.flow(x_augmented, np.zeros(augment_size),  
    batch_size=augment_size, shuffle=False).next()[0]
```

<sup>17</sup> 제가 인터넷에서 찾은 `Iterator`에 대한 가장 쉬운 설명은 이곳(<http://bit.ly/2YORUwZ>)에 있습니다.

```
# 원래 데이터인 x_train에 이미지 보강된 x_augmented를 추가합니다.  
train_X = np.concatenate((train_X, x_augmented))  
train_Y = np.concatenate((train_Y, y_augmented))  
  
print(train_X.shape)
```

#### [OUT]

```
(90000, 28, 28, 1)
```

훈련 데이터의 50%인 30,000개의 이미지를 추가하기 위해 `augment_size = 30000`으로 설정하고, 이미지를 변형할 원본 이미지를 찾기 위해 `np.random.randint()` 함수를 써서 0 ~ 59,999 범위의 정수 중에서 30,000개의 정수를 뽑았습니다. 이때 뽑히는 정수는 중복될 수 있습니다. 중복되지 않는 것을 원한다면 `np.random.randint()` 대신 `np.random.choice()` 함수를 사용하고 `replace` 인수를 `False`로 설정하면 됩니다.

`randidx`는 [2, 25432, 425, …]와 같은 정수로 구성된 넘파이 array이고, 이 array를 이용해 `train_X`에서 각 array의 원소가 가리키는 이미지를 `train_X[randidx]`로 한번에 선택할 수 있습니다. 이렇게 선택한 데이터는 원본 데이터를 참조하는 형태이기 때문에 원본 데이터에 영향을 주지 않기 위해 `copy()` 함수로 안전하게 복사본을 만들어줍니다. 그다음에는 `ImageDataGenerator`의 `flow()` 함수로 30,000개의 새로운 이미지를 생성합니다.

마지막으로 `np.concatenate()` 함수로 훈련 데이터에 보강 이미지를 추가합니다. 최종 출력에서 `train_X.shape`의 첫 번째 차원 수는 90,000이 되어 정상적으로 이미지가 추가된 것을 확인할 수 있습니다.

그럼 이제 예제 6.12의 VGGNet 스타일의 네트워크에 `ImageDataGenerator`로 보강된 훈련 데이터를 학습시켜보겠습니다.

### 예제 6.16 VGGNet style 네트워크 + 이미지 보강학습

#### [IN]

```
model = tf.keras.Sequential([  
    tf.keras.layers.Conv2D(input_shape=(28,28,1), kernel_size=(3,3), filters=32,  
    padding='same', activation='relu'),  
    tf.keras.layers.Conv2D(kernel_size=(3,3), filters=64, padding='same', activation='relu'),  
    tf.keras.layers.MaxPool2D(pool_size=(2,2)),  
    tf.keras.layers.Dropout(rate=0.5),  
    tf.keras.layers.Conv2D(kernel_size=(3,3), filters=128, padding='same', activation='relu'),
```

```

        tf.keras.layers.Conv2D(kernel_size=(3,3), filters=256, padding='valid', activation='relu'),
        tf.keras.layers.MaxPool2D(pool_size=(2,2)),
        tf.keras.layers.Dropout(rate=0.5),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(units=512, activation='relu'),
        tf.keras.layers.Dropout(rate=0.5),
        tf.keras.layers.Dense(units=256, activation='relu'),
        tf.keras.layers.Dropout(rate=0.5),
        tf.keras.layers.Dense(units=10, activation='softmax')
    ])

model.compile(optimizer=tf.keras.optimizers.Adam(),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(train_X, train_Y, epochs=25, validation_split=0.25)

import matplotlib.pyplot as plt
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], 'b-', label='loss')
plt.plot(history.history['val_loss'], 'r--', label='val_loss')
plt.xlabel('Epoch')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], 'g-', label='accuracy')
plt.plot(history.history['val_accuracy'], 'k--', label='val_accuracy')
plt.xlabel('Epoch')
plt.ylim(0.7, 1)
plt.legend()

plt.show()

model.evaluate(test_X, test_Y, verbose=0)

```

## [OUT]

```

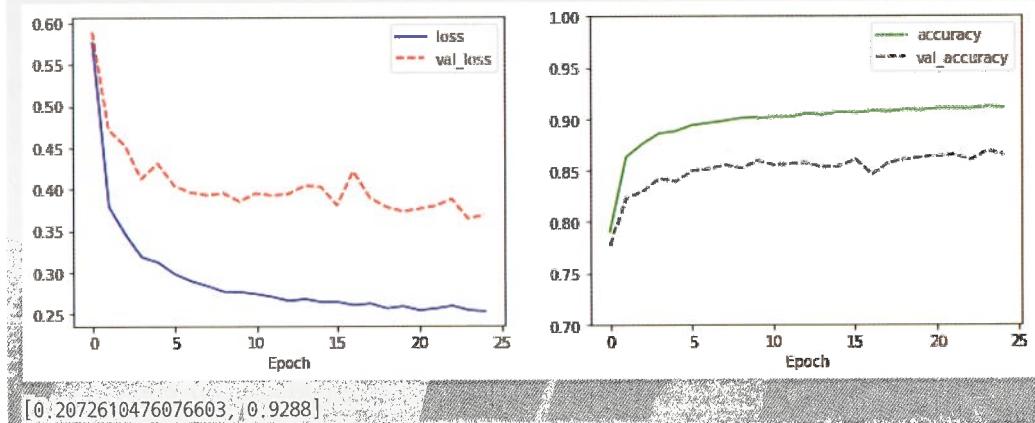
Train on 67500 samples, validate on 22500 samples
Epoch 1/25
67500/67500 [=====] - 15s/219us/sample - loss: 0.5758 - accuracy: 0.7100

```

```

0.7902 - val_loss: 0.5890 - val_accuracy: 0.7776
Epoch 2/25
67500/67500 [=====] - 14s 210us/sample - loss: 0.3788 - accuracy: 0.8630 - val_loss: 0.4716 - val_accuracy: 0.8238
Epoch 3/25
67500/67500 [=====] - 14s 210us/sample - loss: 0.3458 - accuracy: 0.8760 - val_loss: 0.4526 - val_accuracy: 0.8300
Epoch 4/25
67500/67500 [=====] - 14s 209us/sample - loss: 0.3182 - accuracy: 0.8860 - val_loss: 0.4124 - val_accuracy: 0.8428
Epoch 5/25
67500/67500 [=====] - 14s 210us/sample - loss: 0.3118 - accuracy: 0.8881 - val_loss: 0.4308 - val_accuracy: 0.8397
(이하 생략)

```



테스트 데이터에 대한 분류 성적은 92.88%로 92.52%보다 소폭 증가했습니다. val\_accuracy도 증가하는 추세를 보이고 있지 않아서 모델이 아직 과적합되지 않은 것으로 판단되며, 조금 더 학습시키면 성적이 더욱 잘 나올 것으로 기대됩니다.

이렇게 해서 더 많은 레이어를 쌓는 것과 이미지 보강 기법이 컨볼루션 신경망의 분류 성적을 개선할 수 있다는 점을 배웠습니다.

## 6.5 정리

이번 장에서는 딥러닝의 발전을 이끈 컨볼루션 신경망의 특징 추출에 대한 개념과 주요 레이어에 대해 알아봤습니다. 컨볼루션 신경망을 구성하는 주요 레이어로는 컨볼루션 레이어, 풀링 레이어, 드롭아웃 레이어 등이 있습니다.

예제에서는 5장에 나온 Fashion MNIST 데이터세트를 다시 활용해 Dense 레이어만 사용했던 5장과 비교했을 때 성능이 개선된다는 점을 확인했고, 컨볼루션 신경망의 퍼포먼스를 높이는 일반적인 방법인 더 많은 레이어 쌓기와 이미지 보강 기법의 효과를 실제 학습을 통해 확인했습니다.

# 07

## 순환 신경망

순환 신경망(Recurrent Neural Network; RNN)은 지금까지 살펴본 네트워크와는 입력을 받아들이는 방식과 처리하는 방식에서 차이가 있습니다. 순환 신경망은 순서가 있는 데이터를 입력으로 받고, 같은 네트워크를 이용해 변화하는 입력에 대한 출력을 얻어냅니다.

순서가 있는 데이터는 음악, 자연어, 날씨, 주가 등 시간의 흐름에 따라 변화하고 그 변화가 의미를 갖는 데이터입니다. 이번 장에서는 그중 가장 범용적으로 쓰이는 자연어 처리에 순환 신경망을 사용하는 방법을 알아봅니다.

### 7.1 순환 신경망의 구조

지금까지 살펴본 딥러닝 네트워크의 구조를 간략화하면 그림 7.1의 왼쪽처럼 나타낼 수 있습니다. 오른쪽의 순환 신경망은 일반적인 딥러닝 네트워크와 입력  $X$ 를 받아서 출력  $Y$ 를 반환하는 것은 동일하지만, 되먹임 구조를 가지고 있다는 차이점이 있습니다. 여기서 되먹임 구조란 어떤 레이어의 출력을 다시 입력으로 받는 구조를 말합니다.

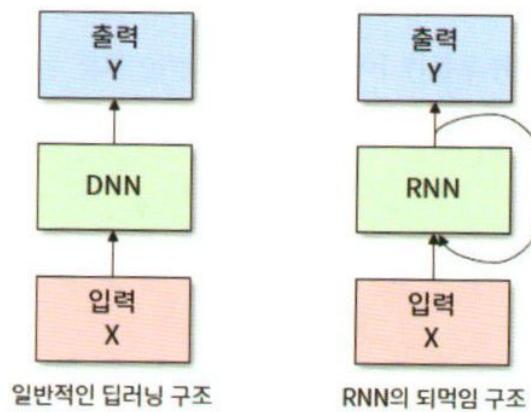


그림 7.1 일반적인 딥러닝 네트워크(DNN)와 순환 신경망(RNN)의 구조 차이

순환 신경망의 구조를 풀어보면 그림 7.2처럼 입력이  $X_1, X_2, X_3$ 으로 변할 때 같은 네트워크를 사용해 출력인  $Y_1, Y_2, Y_3$ 를 반환하고 있음을 확인할 수 있습니다. 이때 꼭 염두에 둬야 할 점은 출력값이 다음 입력을 받을 때의 RNN 네트워크에도 동일하게 전달되고 있다는 것입니다. 즉, RNN 네트워크는 처음에는  $X_1$ 을 입력으로 받고, 그다음에는  $X_2$ 와 이전 단계의 출력인  $Y_1$ , 그다음에는  $X_3$ 과 이전 단계의 출력인  $Y_2$ 를 입력으로 받습니다. 이 과정에서 RNN 네트워크는 동일하게 사용됩니다.

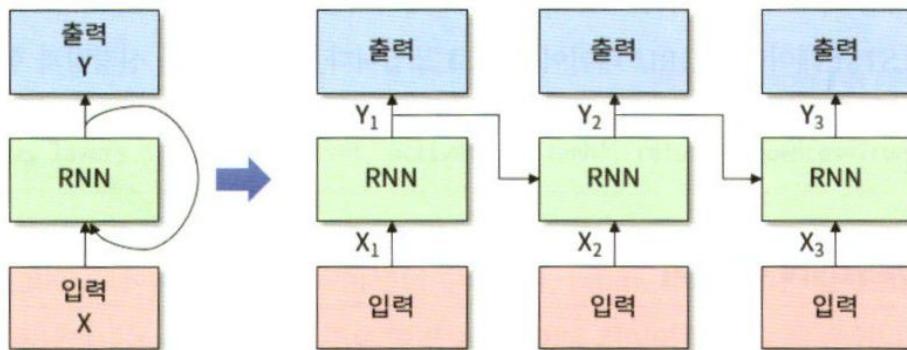


그림 7.2 순환 신경망 구조 풀이

순환 신경망은 입력과 출력의 길이에 제한이 없다는 특징이 있습니다. 따라서 그림 7.3과 같은 다양한 형태의 네트워크를 만드는 것이 가능합니다. 각 네트워크는 이미지를 입력했을 때 이미지에 대한 설명을 생성하는 이미지 설명 생성(Image Captioning), 문장의 긍정/부정을 판단하는 감성 분석(Sentiment Classification), 하나의 언어를 다른 언어로 번역하는 기계 번역(Machine Translation) 등 다양한 용도로 활용됩니다.

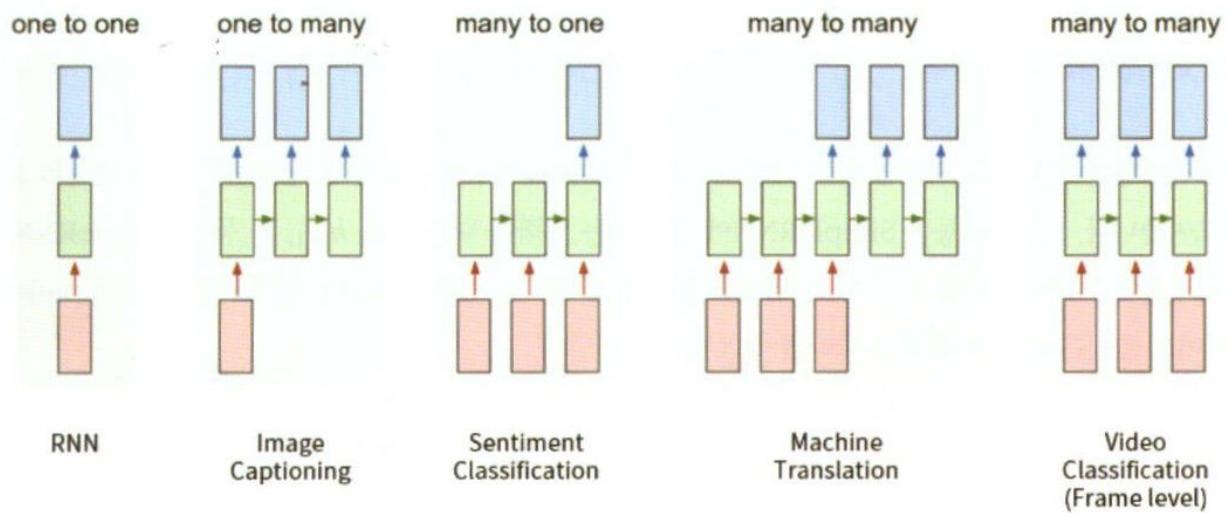


그림 7.3 다양한 형태의 순환 신경망<sup>1</sup>

<sup>1</sup> 출처: 스탠퍼드 cs231n 강의의 10장 'Recurrent Neural Networks' 강의 자료, <https://stanford.io/2GAmLGv>

지금까지 순환 신경망의 구조가 일반적인 딥러닝 신경망과는 다르다는 것을 살펴봤습니다. 그럼 이제 순환 신경망을 구성하는 각 레이어에 대해 알아보겠습니다.

## 7.2 주요 레이어 정리

순환 신경망의 가장 기초적인 레이어는 SimpleRNN 레이어입니다. 실제로는 SimpleRNN 레이어보다 이것의 변종인 LSTM 레이어와 GRU 레이어가 주로 쓰입니다. 그리고 순환 신경망과 함께 자주 쓰이며 자연어 처리를 위해서 꼭 알아둬야 할 임베딩(Embedding) 레이어도 알아보겠습니다.

### 7.2.1 SimpleRNN 레이어

SimpleRNN 레이어는 가장 간단한 형태의 RNN 레이어입니다. 레이어의 구조는 그림 7.4와 같습니다.

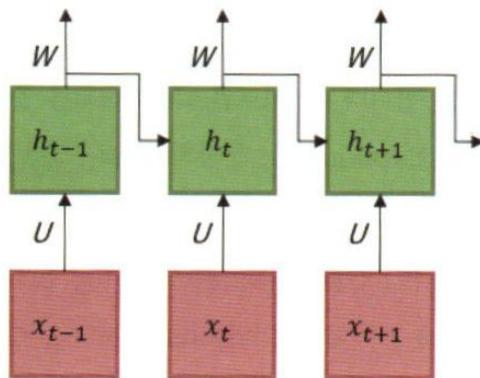


그림 7.4 SimpleRNN 레이어의 구조<sup>2</sup>

순환 신경망은 변화하는 입력을 받기 때문에 그림 7.4에서는 각 단계에서 입력이 변할 때의 계산의 흐름을 보여줍니다.  $x_{t-1}$ ,  $x_t$  등은 SimpleRNN에 들어가는 입력을 나타내고,  $h_{t-1}$ ,  $h_t$  등은 SimpleRNN 레이어의 출력을 나타냅니다.  $U$ 와  $W$ 는 입력과 출력에 곱해지는 가중치입니다. 단계  $t$ 에서의 SimpleRNN 레이어의 출력은 다음 수식으로 나타낼 수 있습니다.

<sup>2</sup> 순환 신경망을 그림이나 수식으로 설명할 때는 보통 편향(bias)을 생략합니다. 이 책에서도 편향은 따로 표시하지 않습니다. 하지만 tf.keras의 계산에서는 편향이 기본적으로 들어가고, 옵션으로 뺄 수 있습니다. SimpleRNN의 편향에 대한 시각화는 이 슬라이드(<http://bit.ly/2TbNFIQ>)의 16페이지에서 확인하실 수 있습니다.

$$h_t = \tanh(U_{x_t} + Wh_{t-1})$$

활성화함수로는 tanh가 쓰입니다. 앞에 나왔던 것처럼 tanh는 실수 입력을 받아  $-1$ 에서  $1$  사이의 출력 값을 반환하는 활성화함수입니다. 활성화함수 자리에 ReLU 같은 다른 활성화함수를 쓸 수도 있습니다.

지금까지 배운 다른 레이어처럼 SimpleRNN 레이어는 tf.keras에서 한 줄로 간단하게 생성할 수 있습니다.

#### 예제 7.1 SimpleRNN 레이어를 생성하는 코드

```
rnn1 = tf.keras.layers.SimpleRNN(units=1, activation='tanh', return_sequences=True)
```

`units`는 SimpleRNN 레이어에 존재하는 뉴런의 수입니다. `return_sequences`는 출력으로 시퀀스 전체를 출력할지 여부를 나타내는 옵션으로서, 주로 여러 개의 RNN 레이어를 쌓을 때 쓰입니다. 이 옵션에 대해서는 예제 7.7에서 자세히 설명하겠습니다.

그럼 간단한 예제를 살펴보겠습니다. 시퀀스를 구성하는 앞쪽 4개의 숫자가 주어졌을 때 그다음에 올 숫자를 예측하는 간단한 “시퀀스 예측 모델”을 만들기 위해 SimpleRNN 레이어를 사용해보겠습니다. 예를 들어,  $[0.0, 0.1, 0.2, 0.3]$ 이라는 연속된 숫자가 주어졌을 때  $[0.4]$ 를 예측하는 네트워크를 만드는 것이 목표입니다. 이렇게 정해진 길이의 시퀀스를 주고 다음 값을 예측하는 문제는 지금까지 배운 Dense 레이어만 사용한 신경망으로도 풀 수 있지만 일단은 SimpleRNN 레이어의 동작을 배우는 것이 목적이기 때문에 SimpleRNN 레이어를 사용해 보겠습니다. 이번 장의 뒤에서는 순환 신경망으로 풀 때 훨씬 효과적인 문제들을 살펴보겠습니다.

예제 7.2에서는 학습에 필요한 데이터를 생성합니다.

#### 예제 7.2 시퀀스 예측 데이터 생성

##### [IN]

```
X = []
Y = []
for i in range(6):
    # [0,1,2,3], [1,2,3,4] 같은 정수의 시퀀스를 만듭니다.
    lst = list(range(i,i+4))
```

# 위에서 구한 시퀀스의 숫자들을 각각 10으로 나눈 다음 저장합니다.

```
# SimpleRNN에 각 타임스텝에 하나씩 숫자가 들어가기 때문에 여기서도 하나씩 분리해서 배열에 저장합니다.
```

```
X.append(list(map(lambda c: [c/10], lst)))
```

```
# 정답에 해당하는 4, 5 등의 정수 역시 앞에서처럼 10으로 나눠서 저장합니다.
```

```
Y.append((i+4)/10)
```

```
X = np.array(X)
Y = np.array(Y)
for i in range(len(X)):
    print(X[i], Y[i])
```

### [OUT]

```
[[0. ]
 [0.1]
 [0.2]
 [0.3]] 0.4
[[0.1]
 [0.2]
 [0.3]
 [0.4]] 0.5
[[0.2]
 [0.3]
 [0.4]
 [0.5]] 0.6
[[0.3]
 [0.4]
 [0.5]
 [0.6]] 0.7
[[0.4]
 [0.5]
 [0.6]
 [0.7]] 0.8
[[0.5]
 [0.6]
 [0.7]
 [0.8]] 0.9
```

그다음으로는 SimpleRNN 레이어를 사용한 네트워크를 정의합니다. 모델 구조는 지금까지 계속 봄은 시퀀셜 모델이고, 출력을 위한 Dense 레이어가 뒤에 추가돼 있습니다.

## [IN]

```
model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(units=10, return_sequences=False, input_shape=[4,1]),
    tf.keras.layers.Dense(1)
])

model.compile(optimizer='adam', loss='mse')
model.summary()
```

## [OUT]

Model: "sequential"

Layer (type)	Output Shape	Param #
simple_rnn_1 (SimpleRNN)	(None, 10)	120
dense (Dense)	(None, 1)	11

Total params: 131

Trainable params: 131

Non-trainable params: 0

예제 7.3의 SimpleRNN 레이어에서 주목해야 할 점은 `input_shape`입니다. 여기서 [4,1]은 각각 `timesteps`, `input_dim`을 나타냅니다. 타임스텝(timesteps)이란 순환 신경망이 입력에 대해 계산을 반복하는 횟수이고, `input_dim`은 입력 벡터의 크기를 나타냅니다. 예제 7.2에서 출력했던 X와 Y는 다음과 같습니다.

```
[[0. ]
 [0.1]
 [0.2]
 [0.3]] 0.4
```

여기서 X는 [1,4,1] 차원의 벡터입니다. 가장 첫 차원은 배치 차원이기 때문에 생략하면 두 번째의 4는 타임스텝, 세 번째의 1은 `input_dim`이 됩니다. 그림으로 나타내면 다음과 같습니다.

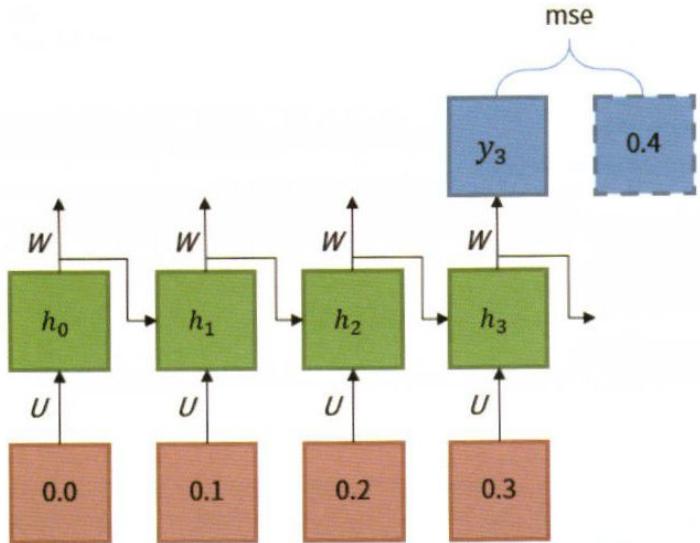


그림 7.5 시퀀스 예측 모델의 구조

시퀀스 예측 모델은 4 타임스텝에 걸쳐 입력을 받고, 마지막에 출력값을 다음 레이어로 반환합니다. 우리가 추가한 Dense 레이어에는 별도의 활성화함수가 없기 때문에  $h_3$ 는 바로  $y_3$ 가 됩니다. 그리고 이 값과 0.4와의 차이가 mse, 즉 평균 제곱 오차(Mean Squared Error)가 됩니다.

네트워크를 정의했으니 훈련을 시켜보겠습니다. 데이터의 수가 적기 때문에 빠르게 훈련이 완료될 것입니다. verbose 값을 0으로 놓으면 훈련 과정에서의 출력이 나오지 않게 할 수 있습니다. 출력을 볼 필요가 없고 훈련만 시키고 싶을 때 유용한 옵션입니다.

#### 예제 7.4 네트워크 훈련 및 결과 확인

##### [IN]

```
model.fit(X, Y, epochs=100, verbose=0)
print(model.predict(X))
```

##### [OUT]

```
[[0.46641627]
 [0.5727007 ]
 [0.66080105]
 [0.72964305]
 [0.7798768 ]
 [0.81313634]]
```

▶ 주어졌을 때 학습된 모델이 시퀀스를 어떻게 예측하는지 확인해보면 얼추 비슷하게 예측하고 있음을 확인할 수 있습니다. 그렇다면 학습 과정에서 본 적이 없는 테스트 데이터를 넣으면 어떨까요?  $x$ 의 범위가 0.0~0.9였으니까 양쪽으로 한 칸씩 더 나간 데이터를 입력해 보겠습니다.

#### 예제 7.5 학습되지 않은 시퀀스에 대한 예측 결과

##### [IN]

```
print(model.predict(np.array([[ [0.6],[0.7],[0.8],[0.9]]])))  
print(model.predict(np.array([[ [-0.1],[0.0],[0.1],[0.2]]])))
```

##### [OUT]

```
[[0.8314607]]  
[[0.3454302]]
```

1.0을 예측하기를 원한 데이터의 출력으로는 0.8314607을, 0.3을 예측하기를 원한 데이터의 출력으로는 0.3454302를 내놓았습니다. 이 결과를 개선하려면 훈련 데이터를 더 많이 넣어주는 것이 좋을 것 같습니다. 하지만 꽤 많은 값을 훈련 데이터에 넣더라도 테스트 데이터에 대한 시퀀스를 정확히 예측하게 하는 것은 쉬운 문제가 아닙니다. 신경망에 일반화된 규칙을 학습시키는 것은 아직 가장 어려운 과제 중 하나입니다.

SimpleRNN 레이어는 순환 신경망의 가장 간단한 형태이면서 빠르게 모델을 만들어볼 때 유용하게 쓰일 수 있습니다. 하지만 실무에서는 SimpleRNN 레이어의 단점을 개선한 LSTM 레이어와 GRU 레이어 등이 많이 쓰입니다.

## 7.2.2 LSTM 레이어

SimpleRNN 레이어에는 한 가지 치명적인 단점이 있습니다. 바로 입력 데이터가 길어질수록, 즉 데이터의 타임스텝이 커질수록 학습 능력이 떨어진다는 점입니다. 이를 장기의존성(Long-Term Dependency) 문제<sup>3</sup>라고 하며, 입력 데이터와 출력 사이의 길이가 멀어질수록 연관 관계가 적어집니다.

<sup>3</sup> Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult, IEEE Transactions on Neural Networks, 5(2), 157 – 166, <http://bit.ly/2YMhAK5>

다.<sup>4</sup> 현재의 답을 얻기 위해 과거의 정보에 의존해야 하는 RNN이지만 과거 시점이 현재와 너무 멀어지면 문제를 풀기 힘들어지는 것입니다.

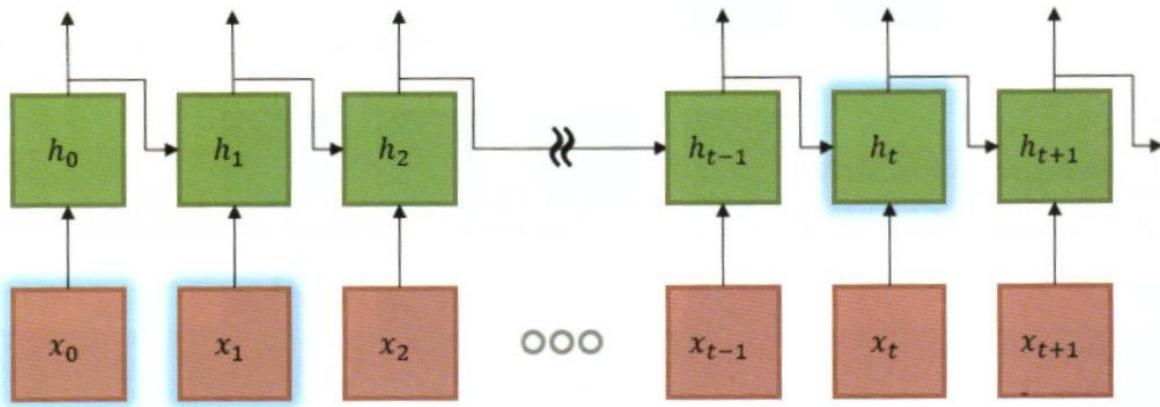


그림 7.6 RNN의 장기의존성 문제. 입력의 길이가 길어지면 먼 거리의 정보를 잘 처리하지 못합니다.

장기의존성 문제를 해결하기 위한 구조로 세프 호흐라이터(Sepp Hochreiter)와 유르겐 슈미트후버(Jürgen Schmidhuber)에 의해 LSTM(Long Short Term Memory)이 1997년에 제안됐습니다.<sup>5</sup> LSTM은 RNN에 비해 복잡한 구조를 가지고 있는데, 가장 큰 특징은 출력 외에 LSTM 셀 사이에서만 공유되는 셀 상태(cell state)를 가지고 있다는 것입니다.

지금까지 배운 SimpleRNN을 셀의 형태로 나타내면 그림 7.7과 같습니다. 타임스텝  $t$ 에서 입력  $x_t$ 와 이전 타임스텝의 출력  $h_{t-1}$ 은 서로 합쳐진(concatenate) 뒤에 활성화함수  $\tanh$ 를 통과하고, 출력  $h_t$ 를 다음 타임스텝의 방향인 오른쪽과 출력 방향인 위쪽으로 내보냅니다. 수식 한 줄로 정리될 만큼 간단한 구조입니다.

4 그림 7.5에서 볼 수 있는 것처럼 SimpleRNN 레이어는 같은 가중치  $W$ 를 반복적으로 사용해서 출력값을 계산합니다. 이런 가중치는 미분을 사용해서 오차를 구하는데, 그림 7.5처럼 화살표로 연결된 관계는 미분 값을 서로 곱해주게 됩니다. 그런데 같은 값을 계속 곱하게 되면 값이 엄청나게 커지는 그레이디언트 폭발(gradient exploding)이나 그레이디언트 소실(gradient vanishing) 문제가 생깁니다. 그리고 활성화함수를  $\tanh$  대신 ReLU로 바꾸면 학습 자체가 불안정해집니다.

5 S. Hochreiter and J. Schmidhuber, Long short-term memory, Neural Computation, 1997. <http://bit.ly/2YsWvsY>

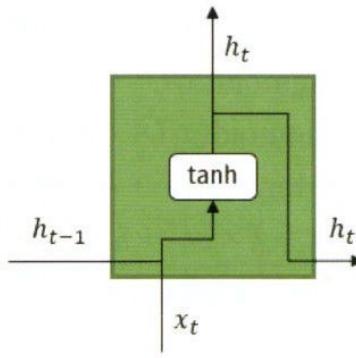


그림 7.7 셀로 나타낸 SimpleRNN 레이어의 계산 흐름

이에 비해 LSTM을 셀의 형태로 나타내면 그림 7.8과 같은데, 꽤 복잡한 모양을 띕니다. 여기서  $c_{t-1}$ 과  $c_t$ 가 바로 셀 상태를 나타내는 기호입니다.

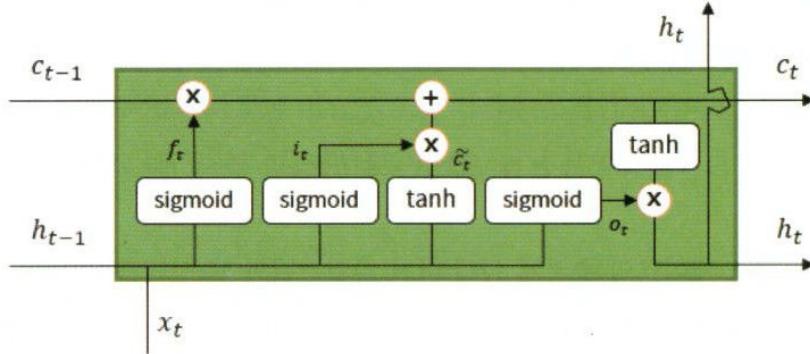


그림 7.8 셀로 나타낸 LSTM 레이어의 계산 흐름

SimpleRNN 셀에서 타임스텝의 방향으로  $h_t$ 만 전달되고 있는 데 비해 LSTM 셀에서는 셀 상태인  $c_t$ 가 평행선을 그리며 함께 전달되고 있습니다. 이처럼 타임스텝을 가로지르며 셀 상태가 보존되기 때문에 장기의존성 문제를 해결할 수 있다는 것이 LSTM의 핵심 아이디어입니다.

LSTM 레이어에는 활성화함수로 tanh 외에 시그모이드 함수가 쓰였습니다. 시그모이드 함수는 항상 0 ~ 1 범위의 출력을 낸다는 것을 이 책의 앞부분에서 설명했습니다. 시그모이드 함수는 이러한 출력의 특성 때문에 정보가 통과하는 게이트 역할을 합니다. 출력이 0이면 입력된 정보가 하나도 통과하지 못하는 것이고, 1이면 100% 통과하게 됩니다.

타임스텝  $t$ 에서의 LSTM 레이어의 출력은 다음 수식으로 나타낼 수 있습니다.

$$\begin{aligned}
 i_t &= \text{sigmoid}(x_t U^i + h_{t-1} W^i) \\
 f_t &= \text{sigmoid}(x_t U^f + h_{t-1} W^f) \\
 o_t &= \text{sigmoid}(x_t U^o + h_{t-1} W^o) \\
 \tilde{c}_t &= \tanh(x_t U^{\tilde{c}} + h_{t-1} W^{\tilde{c}}) \\
 c_t &= f_t \times c_{t-1} + i_t \times \tilde{c}_t \\
 h_t &= \tanh(c_t) \times o_t
 \end{aligned}$$

수식이 꽤 많습니다만 앞에서 언급했던 게이트를 중심으로 설명드리겠습니다.  $U$ 와  $W$ 는 SimpleRNN과 마찬가지로 입력과 출력에 곱해지는 가중치입니다.  $i_t, f_t, o_t$ 는 각각 타임스텝  $t$ 에서의 Input, Forget, Output 게이트를 통과한 출력을 의미합니다.  $\tilde{c}_t$ 는 SimpleRNN에도 존재하던  $x_t$ 와  $h_{t-1}$ 을 각각  $U$ 와  $W$ 에 곱한 뒤에  $\tanh$  활성화함수를 취한 값으로, 셀 상태인  $c^t$ 가 되기 전의 출력값입니다.

마지막의 두 줄은 셀 상태와 LSTM의 출력을 계산하는 가장 중요한 부분입니다. 앞의 게이트에서 계산한 결과에 의해 이 두 값이 결정됩니다.

$$c_t = f_t \times c_{t-1} + i_t \times \tilde{c}_t$$

여기서 셀 상태는 Forget 게이트의 출력에 의해 이전 타임스텝의 셀 상태를 얼마만큼 남길지가 결정되고, 새로 입력된 Input 게이트의 출력과  $\tilde{c}_t$ 를 곱한 값을 더해서 다음 타임스텝의 셀 상태를 만듭니다.

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

LSTM의 출력인  $h_t$ 는 위줄에서 계산된 셀 상태에  $\tanh$  활성화함수를 취한 값을 Output 게이트의 출력에 곱합니다. 이제 Input, Forget, Output 게이트라는 이름이 조금은 더 이해가 되실지 모르겠습니다. 자세한 그림과 함께하는 설명이 필요하다면 구글 브레인(Google Brain)을 거쳐 현재 OpenAI에서 근무 중인 크리스토퍼 올라(Christopher Olah)의 블로그 글을 참고합니다.<sup>6</sup> 이 책에서는 이제 예제 코드와 함께 LSTM의 유용성을 살펴보는 데 집중하겠습니다.

LSTM의 학습 능력을 확인하기 위해 살펴볼 예제 코드는 LSTM을 처음 제안한 논문에 나온 실험 여섯 개 중 다섯 번째인 곱셈 문제(Multiplication problem)입니다. 이 문제는 말 그대로 실수에 대해 곱셈

<sup>6</sup> 원문: <http://bit.ly/33fgudj>, 한글 번역본: <http://bit.ly/2T7Qb4m>

을 하는 문제인데, 고려해야 할 실수의 범위가 100개이고 그중에서 마킹된 두 개의 숫자만 곱해야 한다는 특이한 문제입니다.

마킹 인덱스		1				...		1		
실수	0.1	0.5	0.3	0.8	0.2	...	0.5	0.4	0.3	0.9
100개										
정답	$0.5 \times 0.4 = 0.2$									

그림 7.9 곱셈 문제

그림 7.9에서 마킹 인덱스는 100개의 실수 중 0.5와 0.4에 표시돼 있고, 정답은  $0.5 \times 0.4 = 0.2$ 가 됩니다. 이 문제가 약간 생소할 수 있기 때문에 어느 정도로 어려운 문제인지 감이 안 잡힐 수 있습니다. 그래서 앞에서 배운 SimpleRNN 레이어를 이용한 네트워크로 먼저 문제를 풀어보고, 그다음에 LSTM 레이어를 사용하겠습니다.

먼저 데이터를 만들어야 합니다. 원래 논문에서는 2,560개의 문제를 만들었습니다. 여기서는 3,000개를 만들고, 2,560개를 훈련 데이터와 검증 데이터로 사용하고 나머지 440개는 테스트 데이터로 사용하겠습니다.

#### 예제 7.6 곱셈 문제 데이터 생성

[IN]

```
X = []
Y = []
for i in range(3000):
    # 0 ~ 1 범위의 랜덤한 숫자 100개를 만듭니다.
    lst = np.random.rand(100)
    # 마킹할 숫자 2개의 인덱스를 뽑습니다.
    idx = np.random.choice(100, 2, replace=False)
    # 마킹 인덱스가 저장된 원-핫 인코딩 벡터를 만듭니다.
    zeros = np.zeros(100)
    zeros[idx] = 1
    # 마킹 인덱스와 랜덤한 숫자를 합쳐서 X에 저장합니다.
```

```
X.append(np.array(list(zip(zeros, lst))))  
# 마킹 인덱스가 1인 값만 서로 곱해서 Y에 저장합니다.  
Y.append(np.prod(lst[idx]))  
  
print(X[0], Y[0])
```

### [OUT]

```
[[0.          0.71500918]  
 [0.          0.50462317]  
 [0.          0.79823579]  
 [0.          0.87088728]  
(생략)  
 [0.          0.24426289]  
 [1.          0.00469047]  
 [0.          0.11305018]  
(생략)  
 [1.          0.3036142 ]  
 [0.          0.37056323]  
 [0.          0.3128427 ]  
(생략)  
 [0.          0.58430952]  
 [0.          0.57418694]  
 [0.          0.97540729]] 0.0014240921999715036
```

x와 y의 첫 원소를 출력해보면 x의 첫 번째 열은 마킹 인덱스로, 1은 두 번만 들어가 있습니다. 마킹된 실수인 0.0047과 0.3036을 곱하면 0.0014가 됩니다.

그럼 이제 SimpleRNN 레이어를 이용한 곱셈 문제 모델을 정의해보겠습니다. 이 문제는 앞에서 본 시퀀스 예측 문제보다 약간 어렵기 때문에 레이어의 뉴런 수도 늘리고 SimpleRNN 레이어를 두 층으로 겹친 모델 구조를 사용하겠습니다.

### 예제 7.7 SimpleRNN 레이어를 이용한 곱셈 문제 모델 정의

### [IN]

```
model = tf.keras.Sequential([  
    tf.keras.layers.SimpleRNN(units=30, return_sequences=True, input_shape=[100,2]),  
    tf.keras.layers.SimpleRNN(units=30),  
    tf.keras.layers.Dense(1)  
])
```

```
model.compile(optimizer='adam', loss='mse')
model.summary()
```

## [OUT]

Model: "sequential\_1"

Layer (type)	Output Shape	Param #
simple_rnn_2 (SimpleRNN)	(None, 100, 30)	990
simple_rnn_3 (SimpleRNN)	(None, 30)	1830
dense_1 (Dense)	(None, 1)	31

Total params: 2,851

Trainable params: 2,851

Non-trainable params: 0

RNN 레이어를 겹치기 위해 첫 번째 SimpleRNN 레이어에서 `return_sequences=True`로 설정된 것을 확인할 수 있습니다. `return_sequences`는 레이어의 출력을 다음 레이어로 그대로 넘겨주게 됩니다. 네트워크의 구조는 그림 7.10과 같습니다.

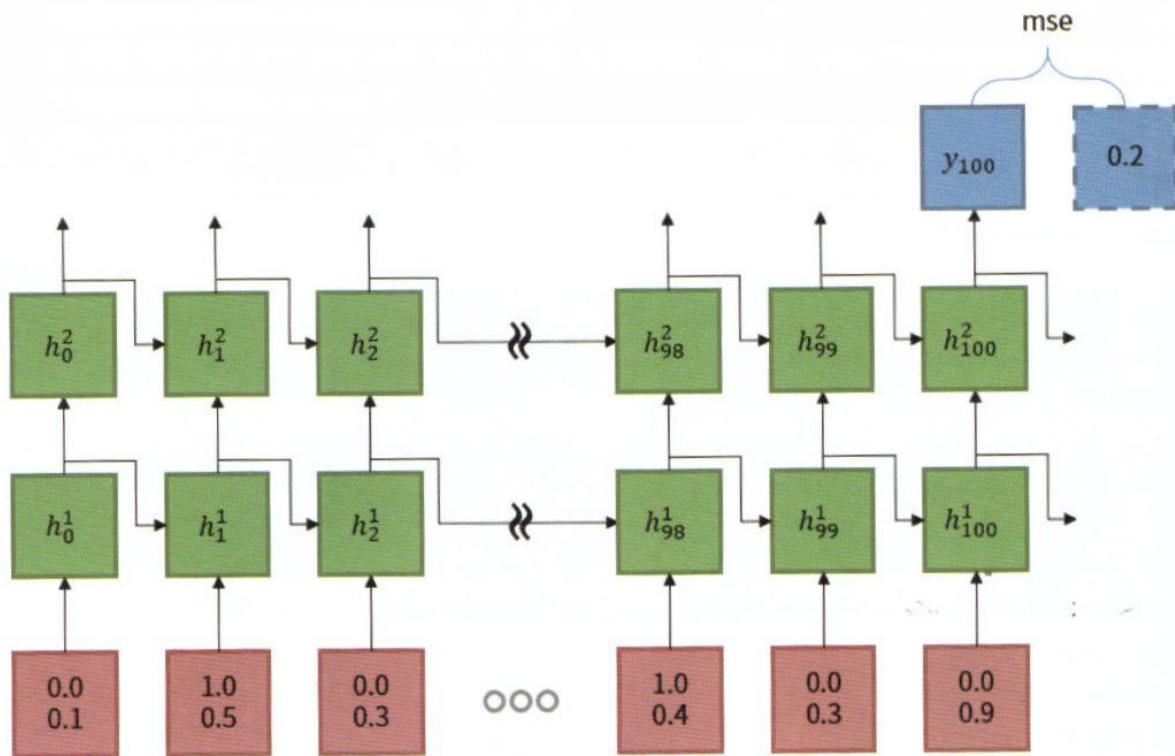


그림 7.10 겹쳐진 SimpleRNN 레이어 구조

$h_0^1$ 이라고 돼 있는 부분에서 윗첨자에 해당하는 오른쪽 위의 숫자 1은 첫 번째 레이어를 의미합니다. 첫 번째 SimpleRNN 레이어는 모든 출력을 다음 레이어로 넘기기 때문에 두 번째 SimpleRNN 레이어도 각 타임스텝에 대해 아래쪽과 옆에서 오는 양방향의 입력을 정상적으로 받을 수 있습니다. 두 번째 SimpleRNN 레이어는 `return_sequences` 인수가 지정돼 있지 않기 때문에 기본값인 `False`가 돼서 마지막 계산값만 출력으로 넘기고, 마지막의 Dense 레이어의 출력과 정답과의 평균 제곱 오차를 비교하고 이 오차를 줄이는 방향으로 네트워크를 학습시키게 됩니다.

그럼 실제로 학습을 시켜보겠습니다. 순환신경망은 앞에서 배운 컨볼루션 신경망보다 학습 시간이 오래 걸리는 편이기 때문에 하드웨어 가속기를 GPU로 바꾸는 것이 좋습니다. 상단 메뉴에서 ‘런타임’ → ‘런타임 유형 변경’ → ‘하드웨어 가속기’를 차례로 선택한 후 ‘GPU’를 선택합니다.

#### 예제 7.8 SimpleRNN 네트워크 학습

##### [IN]

```
X = np.array(X)
Y = np.array(Y)
# 2560개의 데이터만 학습시킵니다. 검증 데이터는 20%로 지정합니다.
history = model.fit(X[:2560], Y[:2560], epochs=100, validation_split=0.2)
```

**[OUT]**

```
Train on 2048 samples, validate on 512 samples
Epoch 1/100
2048/2048 [=====] - 19s 9ms/sample - loss: 0.0987 - val_loss: 0.0544
Epoch 2/100
2048/2048 [=====] - 18s 9ms/sample - loss: 0.0530 - val_loss: 0.0514
Epoch 3/100
2048/2048 [=====] - 19s 9ms/sample - loss: 0.0529 - val_loss: 0.0567
Epoch 4/100
2048/2048 [=====] - 19s 9ms/sample - loss: 0.0531 - val_loss: 0.0512
Epoch 5/100
2048/2048 [=====] - 19s 9ms/sample - loss: 0.0520 - val_loss: 0.0504
(생략)
Epoch 96/100
2048/2048 [=====] - 18s 9ms/sample - loss: 0.0252 - val_loss: 0.0701
Epoch 97/100
2048/2048 [=====] - 18s 9ms/sample - loss: 0.0250 - val_loss: 0.0684
Epoch 98/100
2048/2048 [=====] - 18s 9ms/sample - loss: 0.0259 - val_loss: 0.0682
Epoch 99/100
2048/2048 [=====] - 19s 9ms/sample - loss: 0.0246 - val_loss: 0.0708
Epoch 100/100
2048/2048 [=====] - 19s 9ms/sample - loss: 0.0242 - val_loss: 0.0748
```

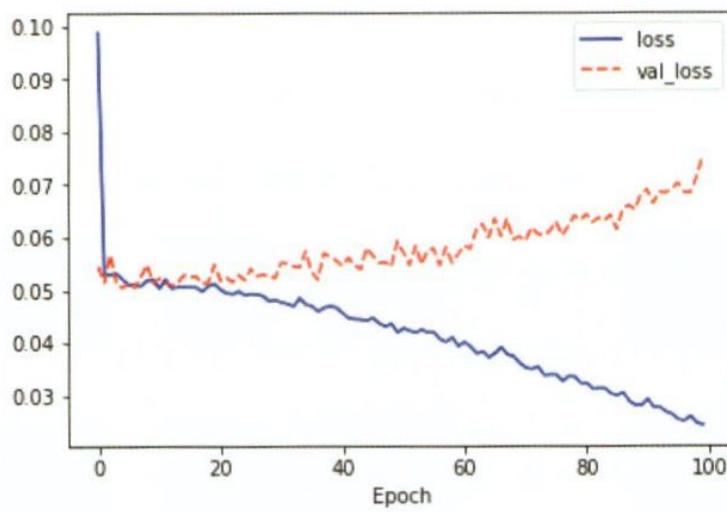
훈련 데이터의 손실인 loss는 점차 감소하지만 검증 데이터의 손실인 val\_loss는 감소하지 않고 오히려 증가하는 듯합니다. 경향을 직관적으로 파악하기 위해 history 변수에 저장된 값으로 그래프를 그려보겠습니다.

예제 7.9 SimpleRNN 네트워크 학습 결과 확인

**[IN]**

```
import matplotlib.pyplot as plt
plt.plot(history.history['loss'], 'b-', label='loss')
plt.plot(history.history['val_loss'], 'r--', label='val_loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()
```

## [OUT]



학습 결과는 전형적인 과적합 그래프를 보여줍니다. 그렇다면 테스트 데이터에 대한 예측은 어떨까요? 논문에서는 오차가 0.04 이상일 때 오답으로 처리했습니다. 여기서도 이 조건으로 예측의 정확도를 체크해보겠습니다.

### 예제 7.10 테스트 데이터에 대한 예측 정확도 확인

## [IN]

```
model.evaluate(X[2560:], Y[2560:])
prediction = model.predict(X[2560:2560+5])
# 5개 테스트 데이터에 대한 예측을 표시합니다.
for i in range(5):
    print(Y[2560+i], '\t', prediction[i][0], '\tdiff:', abs(prediction[i][0] - Y[2560+i]))

prediction = model.predict(X[2560:])
fail = 0
for i in range(len(prediction)):
    # 오차가 0.04 이상이면 오답입니다.
    if abs(prediction[i][0] - Y[2560+i]) > 0.04:
        fail += 1
print('correctness:', (440 - fail) / 440 * 100, '%')
```

## [OUT]

```
440/440 [=====] - 1s 2ms/sample - loss: 0.0805
0.034887773363423434      0.37500668      diff: 0.34011890235679143
```

```

0.00034131119754715394 0.012471177 diff: 0.012129865486355586
0.4893153311074549 0.4436293 diff: 0.04568603647358954
0.153814953074304 0.32798985 diff: 0.1741748933936678
0.22067859331833786 0.13175683 diff: 0.088921766083116
correctness: 9.318181818181818 %

```

먼저 전체에 대한 평가(evaluate)로 0.0805의 loss가 나왔습니다. 위에서 본 100번째 에포크의 val\_loss인 0.0748보다도 높은 값으로, 네트워크가 학습 과정에서 한번도 못 본 테스트 데이터에 대해서는 잘 예측하지 못합니다. 5개의 테스트 데이터에 대한 샘플은 오차가 0.01에서 0.34까지 다양하게 나타나며, 가장 중요한 정확도는 9.32%로 나옵니다.

그렇다면 LSTM 레이어는 어떨까요? 이 문제를 풀기 위한 시퀀셜 모델을 정의해보겠습니다.

#### 예제 7.11 LSTM 레이어를 이용한 곱셈 문제 모델 정의

##### [IN]

```

model = tf.keras.Sequential([
    tf.keras.layers.LSTM(units=30, return_sequences=True, input_shape=[100,2]),
    tf.keras.layers.LSTM(units=30),
    tf.keras.layers.Dense(1)
])

model.compile(optimizer='adam', loss='mse')
model.summary()

```

##### [OUT]

Model: "sequential\_2"

Layer (type)	Output Shape	Param #
unified_lstm (UnifiedLSTM)	(None, 100, 30)	3960
unified_lstm_1 (UnifiedLSTM)	(None, 30)	7320
dense_2 (Dense)	(None, 1)	31

Total params: 11,311

Trainable params: 11,311

Non-trainable params: 0

예제 7.7과 예제 7.11의 차이점은 SimpleRNN을 LSTM으로 바꾼 것 외에는 없습니다. 모두 동일한 조건에서 레이어만 바꿨을 때 어떤 차이가 나는지 알아보는 것이 이 실험의 목적이기 때문입니다.

네트워크의 학습 코드도 동일합니다.

### 예제 7.12 LSTM 네트워크 학습

#### [IN]

```
X = np.array(X)
Y = np.array(Y)
history = model.fit(X[:2560], Y[:2560], epochs=100, validation_split=0.2)
```

#### [OUT]

```
Train on 2048 samples, validate on 512 samples
Epoch 1/100
2048/2048 [=====] - 6s 3ms/sample - loss: 0.0526 - val_loss: 0.0508
Epoch 2/100
2048/2048 [=====] - 2s 1ms/sample - loss: 0.0484 - val_loss: 0.0507
Epoch 3/100
2048/2048 [=====] - 2s 1ms/sample - loss: 0.0482 - val_loss: 0.0514
Epoch 4/100
2048/2048 [=====] - 2s 1ms/sample - loss: 0.0481 - val_loss: 0.0513
Epoch 5/100
2048/2048 [=====] - 2s 1ms/sample - loss: 0.0481 - val_loss: 0.0516
(생략)
Epoch 96/100
2048/2048 [=====] - 2s 1ms/sample - loss: 6.6581e-04 - val_loss:
6.9919e-04
Epoch 97/100
2048/2048 [=====] - 2s 1ms/sample - loss: 5.8539e-04 - val_loss:
7.7874e-04
Epoch 98/100
2048/2048 [=====] - 2s 1ms/sample - loss: 5.9506e-04 - val_loss:
5.3631e-04
Epoch 99/100
2048/2048 [=====] - 2s 1ms/sample - loss: 0.0012 - val_loss: 0.0015
Epoch 100/100
2048/2048 [=====] - 2s 1ms/sample - loss: 6.5871e-04 - val_loss:
5.9089e-04
```

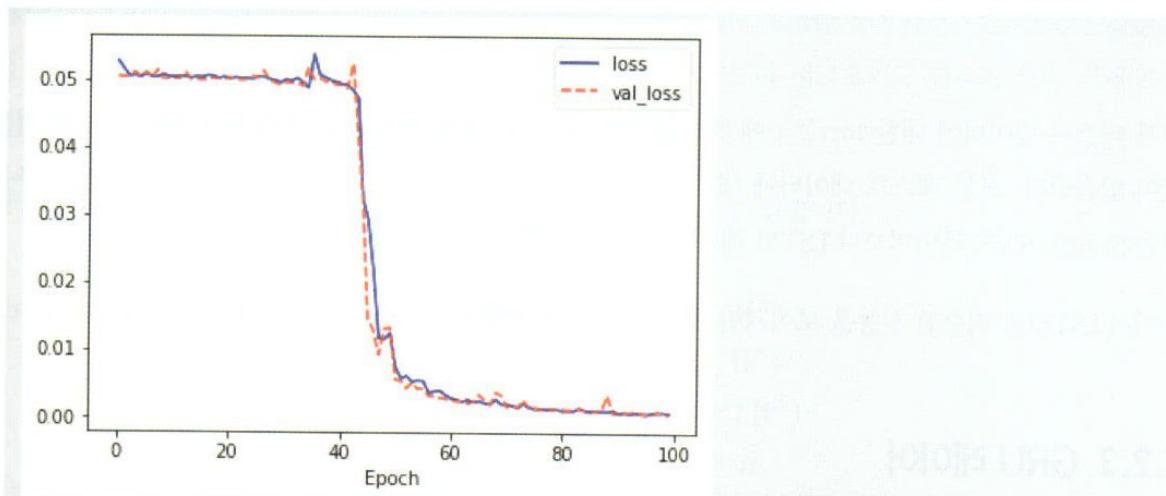
처음에는 loss와 val\_loss가 잘 줄어들지 않는 듯하지만 나중에는 두 값 모두 매우 작아져서 과학적 표기법을 써야 할 정도가 됩니다. 예제 7.13에서 그라프로 학습 결과를 확인해볼 수 있습니다.

#### 예제 7.13 LSTM 네트워크의 학습 결과 확인

##### [IN]

```
import matplotlib.pyplot as plt
plt.plot(history.history['loss'], 'b-', label='loss')
plt.plot(history.history['val_loss'], 'r--', label='val_loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()
```

##### [OUT]



loss와 val\_loss는 40에포크를 넘어가면서 매우 가파르게 줄어들어 0에 가까워집니다. val\_loss는 변동폭이 loss보다 크지만 전체적으로는 계속 감소하는 경향을 보입니다. 학습은 매우 잘 된 것 같습니다. 실제로 테스트 데이터에 대해 얼마나 정확하게 값을 예측하는지도 확인해보겠습니다.

#### 예제 7.14 테스트 데이터에 대한 예측 정확도 확인

##### [IN]

```
model.evaluate(X[2560:], Y[2560:])
prediction = model.predict(X[2560:2560+5])
for i in range(5):
    print(Y[2560+i], '\t', prediction[i][0], '\tdiff:', abs(prediction[i][0] - Y[2560+i]))
```

```

prediction = model.predict(X[2560:])
cnt = 0
for i in range(len(prediction)):
    if abs(prediction[i][0] - Y[2560+i]) > 0.04:
        cnt += 1
print('correctness:', (440 - cnt) / 440 * 100, '%')

```

#### [OUT]

```

440/440 [=====] - 0s 265us/sample - loss: 6.0522e-04
0.03488773363423434 0.045945946 diff: 0.01105817276375285
0.00034131119754715394 0.013404831 diff: 0.013063520092697902
0.4893153311074549 0.5186676 diff: 0.02935224758974969
0.153814953074304 0.14587536 diff: 0.007939588532296565
0.22067859331833786 0.21842426 diff: 0.002254332702035372
correctness: 94.77272727272728 %

```

먼저 테스트 데이터에 대한 loss는 0에 가까운 값이 나오고, 다섯 개의 샘플에 대한 오차도 0.04를 넘는 값이 없습니다. 모든 테스트 데이터에 대한 정확도는 94.8%로 거의 95%에 가깝습니다. 이 문제를 푸는데는 SimpleRNN 레이어보다 LSTM 레이어가 훨씬 적합하다는 것을 확인할 수 있습니다.

이어서 LSTM과 비슷한 성능을 보이지만 좀 더 구조가 간단한 GRU 레이어에 대해서도 소개하겠습니다.

### 7.2.3 GRU 레이어

GRU(Gated Recurrent Unit) 레이어는 뉴욕대학교의 조경현 교수 등이 제안한 구조입니다.<sup>7</sup> 조경현 교수는 앞에서 나온 장기의존성에 관한 논문을 쓴 요슈아 벤지오 교수의 제자입니다. GRU 레이어는 LSTM 레이어와 비슷한 역할을 하지만 구조가 더 간단하기 때문에 계산상의 이점이 있고, 어떤 문제에서는 LSTM 레이어보다 좋은 성능을 보이기도 합니다.<sup>8</sup>

셀로 나타낸 GRU 레이어의 계산 흐름은 그림 7.11과 같은데, LSTM보다는 단순한 모습입니다.

<sup>7</sup> Cho, K., van Merriënboer, B., Gulcehre, C., Bougares, F., Schwenk, H., and Bengio, Y. (2014a).

Learning phrase representations using RNN encoder–decoder for statistical machine translation,

In Proceedings of the Empirical Methods in Natural Language Processing, EMNLP 2014, <https://arxiv.org/abs/1406.1078>

<sup>8</sup> Chung, Junyoung, Gulcehre, Caglar, Cho, KyungHyun, and Bengio, Yoshua, Empirical evaluation of gated recurrent neural networks on sequence modeling, <https://arxiv.org/abs/1412.3555>

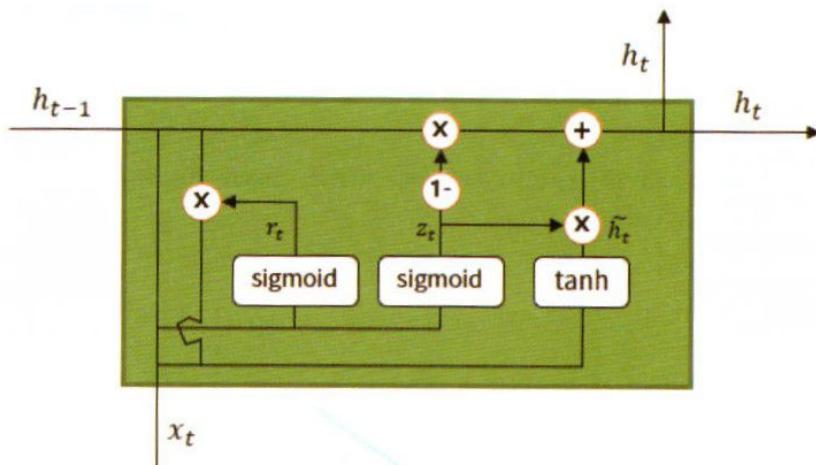


그림 7.11 셀로 나타낸 GRU 레이어의 계산 흐름

LSTM과의 가장 큰 차이점은 셀 상태가 보이지 않는다는 것입니다. GRU 레이어에는 셀 상태가 없는 대신  $h_t$ 가 비슷한 역할을 합니다. GRU 레이어에는 LSTM 레이어보다 시그모이드 함수가 하나 적게 쓰였는데, 이것은 게이트의 수가 하나 줄어들었다는 것을 의미합니다.

타임스텝  $t$ 에서의 GRU 레이어의 출력은 다음 수식으로 나타낼 수 있습니다.

$$\begin{aligned} z_t &= \text{sigmoid}(x_t U^z + h_{t-1} W^z) \\ r_t &= \text{sigmoid}(x_t U^r + h_{t-1} W^r) \\ \tilde{h}_t &= \tanh(x_t U^{\tilde{h}} + (h_{t-1} \times r_t) W^{\tilde{h}}) \\ h_t &= (1 - z_t) \times h_{t-1} + z_t \times \tilde{h}_t \end{aligned}$$

$r_t$ 는 Reset 게이트,  $z_t$ 는 Update 게이트를 통과한 출력입니다. Reset 게이트를 통과한 출력  $r_t$ 은 이전 타임스텝의 출력인  $h_t$ 에 곱해지기 때문에 이전 타임스텝의 정보를 얼마나 남길지를 결정하는 정도라고 생각할 수 있습니다. Update 게이트의 출력  $z_t$ 는 LSTM의 Input과 Forget 게이트의 출력의 역할을 동시에 수행하는 듯한 형태입니다. 위의 수식 4번째 줄에서  $\tanh$ 을 통과한  $\tilde{h}_t$ 와 이전 타임스텝의 출력인  $h_{t-1}$ 은  $z_t$  값에 따라 최종 출력에서 각각 어느 정도의 비율을 점유할지 결정되며, 그 결과로 이전 타임스텝의 정보를 얼마나 남길지가 결정됩니다.

$$h_t = (1 - z_t) \times \tilde{h}_t + z_t \times h_{t-1}$$

그럼 이제 GRU 레이어는 앞에서 SimpleRNN 레이어와 LSTM 레이어로 풀었던 곱셈 문제를 얼마나 잘 풀 수 있는지 확인해보겠습니다.

## [IN]

```
model = tf.keras.Sequential([
    tf.keras.layers.GRU(units=30, return_sequences=True, input_shape=[100,2]),
    tf.keras.layers.GRU(units=30),
    tf.keras.layers.Dense(1)
])

model.compile(optimizer='adam', loss='mse')
model.summary()
```

## [OUT]

```
Model: "sequential_3"

Layer (type)          Output Shape         Param #
unified_gru (UnifiedGRU)    (None, 100, 30)      3060
unified_gru_1 (UnifiedGRU)   (None, 30)          5580
dense_3 (Dense)           (None, 1)            31

Total params: 8,671
Trainable params: 8,671
Non-trainable params: 0
```

LSTM만 GRU로 바꿔서 간단히 모델을 정의할 수 있습니다. GRU 레이어를 사용한 네트워크의 파라미터 수는 LSTM 레이어의 파라미터 수보다 적습니다. 표 7.1에 세 가지 레이어를 사용한 네트워크의 파라미터 수를 정리했습니다. GRU 레이어를 사용한 네트워크의 파라미터 수는 LSTM 레이어 네트워크보다 약 23.3% 정도 감소한 수치입니다.

표 7.1 곱셈 문제를 풀기 위한 네트워크의 파라미터 수

SimpleRNN	LSTM	GRU
2,851	11,311	8,671

그럼 이제 실제로 학습시켜보겠습니다. 학습 코드도 앞의 예제와 동일합니다.

### 예제 7.16 GRU 네트워크 학습

#### [IN]

```
X = np.array(X)
Y = np.array(Y)
history = model.fit(X[:2560], Y[:2560], epochs=100, validation_split=0.2)
```

#### [OUT]

```
Train on 2048 samples, validate on 512 samples
Epoch 1/100
2048/2048 [=====] - 3s 1ms/sample - loss: 0.0565 - val_loss: 0.0515
Epoch 2/100
2048/2048 [=====] - 2s 1ms/sample - loss: 0.0483 - val_loss: 0.0508
Epoch 3/100
2048/2048 [=====] - 2s 1ms/sample - loss: 0.0483 - val_loss: 0.0507
Epoch 4/100
2048/2048 [=====] - 2s 1ms/sample - loss: 0.0481 - val_loss: 0.0511
Epoch 5/100
2048/2048 [=====] - 2s 1ms/sample - loss: 0.0480 - val_loss: 0.0508
(생략)
Epoch 96/100
2048/2048 [=====] - 2s 1ms/sample - loss: 1.6273e-04 - val_loss:
2.3228e-04
Epoch 97/100
2048/2048 [=====] - 2s 1ms/sample - loss: 1.7745e-04 - val_loss:
1.0128e-04
Epoch 98/100
2048/2048 [=====] - 2s 1ms/sample - loss: 1.2225e-04 - val_loss:
1.2500e-04
Epoch 99/100
2048/2048 [=====] - 2s 1ms/sample - loss: 1.8092e-04 - val_loss:
1.9567e-04
Epoch 100/100
2048/2048 [=====] - 2s 1ms/sample - loss: 1.4453e-04 - val_loss:
1.4514e-04
```

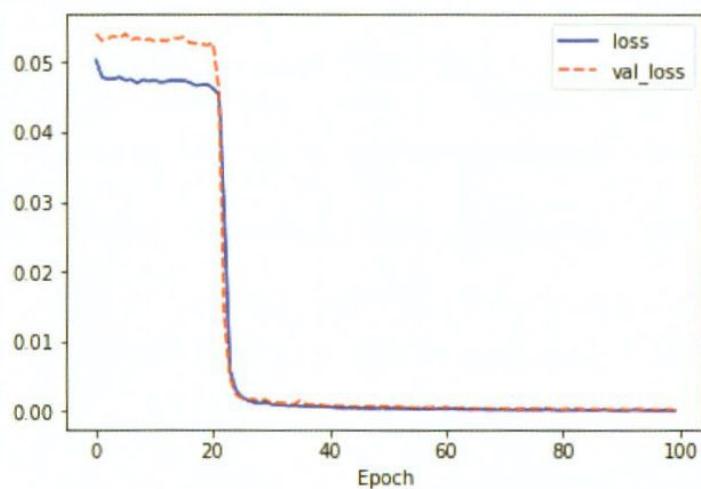
100번째 에포크가 되면 loss와 val\_loss가 LSTM 레이어를 사용했을 때만큼 작아진 것을 확인할 수 있습니다.

#### 예제 7.17 GRU 네트워크의 학습 결과 확인

##### [IN]

```
import matplotlib.pyplot as plt  
plt.plot(history.history['loss'], 'b-', label='loss')  
plt.plot(history.history['val_loss'], 'r--', label='val_loss')  
plt.xlabel('Epoch')  
plt.legend()  
plt.show()
```

##### [OUT]



LSTM 네트워크의 loss와 val\_loss가 약 40에포크 정도에서 가파르게 줄어들었던 것에 비해 GRU 네트워크는 20에포크 정도에서 값이 줄어들고, 값의 변화도 더 안정적입니다.

#### 예제 7.18 테스트 데이터에 대한 예측 정확도 확인

##### [IN]

```
model.evaluate(X[2560:], Y[2560:])  
prediction = model.predict(X[2560:2560+5])  
for i in range(5):  
    print(Y[2560+i], '\t', prediction[i][0], '\tdiff:', abs(prediction[i][0] - Y[2560+i]))
```

```

prediction = model.predict(X[2560:])
cnt = 0
for i in range(len(prediction)):
    if abs(prediction[i][0] - Y[2560+i]) > 0.04:
        cnt += 1
print('correctness:', (440 - cnt) / 440 * 100, '%')

```

## [OUT]

```

440/440 [=====] - 0s 257us/sample - loss: 2.1656e-04
0.5868979267885701      0.59347934      diff: 0.006581408519504839
0.05878754410344112      0.060995653     diff: 0.002208109168233989
0.28405740032203625      0.2864653     diff: 0.0024078868865208825
0.0031702546170603993    0.0039648563    diff: 0.0007946016499150036
0.49710513844098114      0.49417952      diff: 0.0029256214102652
correctness: 98.86363636363636 %

```

정확도는 98.9%로 거의 99%에 가까운 값이 나왔습니다. 이 문제는 LSTM 레이어보다 GRU 레이어로 더 잘 풀리는 문제인 것 같습니다. 다만 딥러닝의 특성상 시행할 때마다 결과가 달라질 수 있기 때문에 정확한 결과를 얻으려면 여러 번 실험해서 평균을 낸 값을 사용하는 것이 좋습니다.

GRU 레이어는 LSTM 레이어보다 적은 파라미터를 가지지만 비슷한 기능을 하고, 곱셈 문제를 포함한 일부 문제에서는 더 좋은 성능을 낼 수도 있다는 것을 확인했습니다.

다음으로는 자연어 처리에서 빼놓을 수 없고 자연어를 다루는 순환 신경망에서 꼭 등장하는 임베딩 레이어에 대해 알아보겠습니다.

## 7.2.4 임베딩 레이어

임베딩 레이어(Embedding Layer)는 자연어를 수치화된 정보로 바꾸기 위한 레이어입니다.

자연어는 시간의 흐름에 따라 정보가 연속적으로 이어지는 시퀀스 데이터입니다. 이미지를 픽셀 단위로 잘게 쪼갤 수 있듯이 자연어도 정보를 잘게 쪼갤 수 있습니다. 영어는 문자(character), 한글은 문자를 넘어 자소 단위로도 쪼갤 수 있습니다. 이보다 더 큰 단위로는 띄어쓰기 단위인 단어가 있습니다. 또 이 방법들과 다르게 몇 개의 문자를 묶어서 파악하려는 n-gram 기법이 있습니다. 예를 들어, "This is it."이라는 문장을 3개의 문자를 묶은 3-gram으로 나타내면 ["Thi", "his", "is ", "s i", " is", "is ",

"s i", " it", "it."]이라는 배열로 나타낼 수 있습니다. 딥러닝 기법이 발달한 이후로는 n-gram보다 단어나 문자 단위의 자연어 처리가 많이 쓰이고 있습니다.<sup>9</sup>



그림 7.12 2018년 8월 ~ 2019년 8월 사이의 구글 트렌드에서의 word embedding과 n-gram 관심도 비교

임베딩 레이어보다 좀 더 쉬운 기법은 자연어를 구성하는 단위에 대해 정수 인덱스(index)를 저장하는 방법입니다. 단어를 기반으로 정수 인덱스를 저장하는 예를 들어보겠습니다. “This is a big cat.”이라는 문장에 대해 정수 인덱스를 저장하면 처음 나오는 단어부터 인덱스를 저장합니다.

표 7.2 단어 기반 정수 인덱스의 예<sup>10</sup>

단어	인덱스
this	0
is	1
a	2
big	3
cat	4

9 페이스북에서 발표한 FastText에서는 단어 기반의 Word2Vec과 n-gram 기법을 함께 사용했습니다.

10 표 7.2의 단어들에서는 대문자를 소문자로 바꿨고 구두점(.)을 생략했습니다. 영어에서 자연어를 처리할 때는 이렇게 대문자를 소문자로 바꾸기, 문장부호 제거, 자주 나오지만 별 의미가 없는 단어(stopword)를 제거하는 등의 과정을 거칩니다.

표 7.2에 의해 원래의 문장은 [0,1,2,3,4]라는 새로운 수치화된 데이터로 변환될 수 있습니다. 이때 “This is big.”이라는 새로운 문장도 [0,1,3]이라는 데이터로 바꿀 수 있습니다. 학습을 위해 신경망에 넣을 데이터로 변환할 때는 5장 ‘분류’에서 배운 원-핫 인코딩을 이용해 단어의 인덱스에 해당하는 원소만 1이고 나머지는 0인 배열로 바꿉니다. 이때 원-핫 인코딩 배열의 두 번째 차원의 크기는 단어의 수와 같습니다. 아래 예시에서 두 번째 차원의 크기는 5입니다.

---

```
[0,1,2,3,4] -> [[1,0,0,0,0],
[0,1,0,0,0],
[0,0,1,0,0],
[0,0,0,1,0],
[0,0,0,0,1]]
[0,1,3] -> [[1,0,0,0,0],
[0,1,0,0,0],
[0,0,0,1,0]]
```

---

인덱스를 사용하는 원-핫 인코딩 방식의 단점은 사용하는 메모리의 양에 비해 너무 적은 정보량을 표현하는 것입니다. 또 한 가지 단점은 인덱스에 저장된 단어의 수가 많아질수록 원-핫 인코딩 배열의 두 번째 차원의 크기도 그에 비례해서 늘어나기 때문에 이 데이터가 차지하는 메모리의 양이 더욱 늘어나게 된다는 것입니다.

원-핫 인코딩 방식에 비해 임베딩 레이어는 한정된 길이의 벡터로 자연어의 구성 단위인 자소, 문자, 단어, n-gram 등을 표현할 수 있습니다.

**표 7.3** 단어 기반 정수 인덱스, 임베딩의 예

단어	인덱스	원-핫 인코딩	임베딩
this	0	[1,0,0,0,0]	[0,1,0,4,0,4,0,0,0,1]
is	1	[0,1,0,0,0]	[0,1,0,2,0,8,0,0,0,1]
a	2	[0,0,1,0,0]	[0,2,0,5,0,0,0,6,0,3]
big	3	[0,0,0,1,0]	[0,8,0,2,0,4,0,8,0,8]
cat	4	[0,0,0,0,1]	[0,4,0,2,0,6,0,9,0,9]

표 7.3에서 임베딩 부분에는 크기 5의 벡터에 임의의 실수를 넣었습니다. 실수는 연속적이기 때문에 임베딩은 이론상 무한대의 단어를 표현할 수 있습니다. 물론 임베딩의 차원 수를 늘리면 표현의 품질은 더 좋아집니다. 보통 임베딩 차원으로는 200에서 500 사이를 사용합니다.<sup>11</sup>

임베딩 레이어가 좋지만 정수 인덱스가 쓸모없는 것은 아닙니다. 임베딩 레이어는 정수 인덱스를 단어 임베딩으로 바꾸는 역할을 하기 때문에 정수 인덱스는 임베딩 레이어의 입력이 됩니다. 정수 인덱스가 단어 임베딩으로 바뀌는 계산 과정은 그림 7.13과 같습니다.

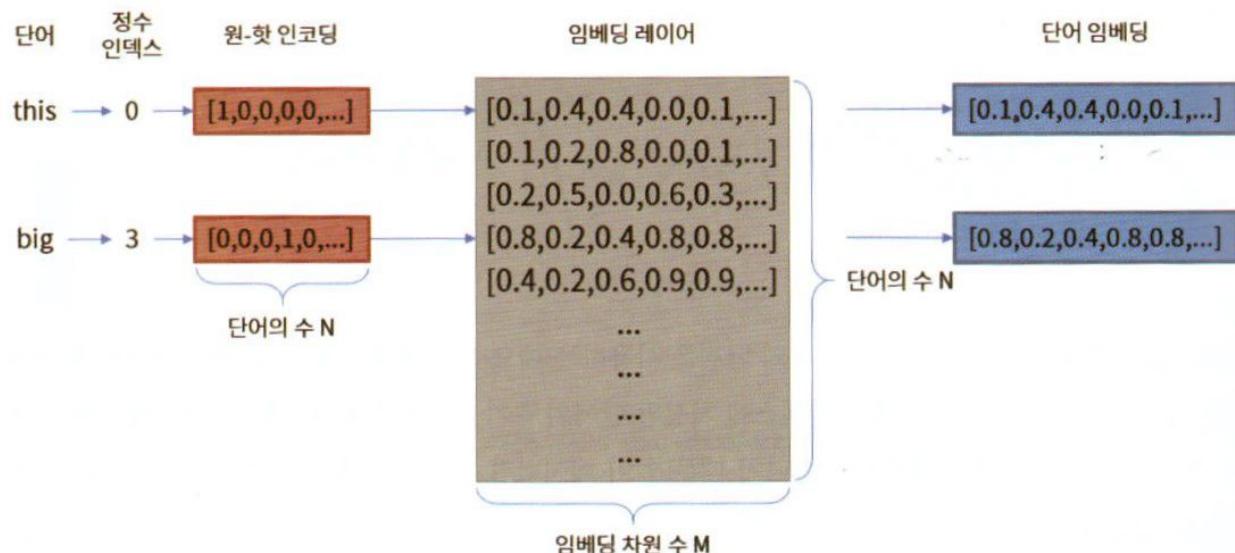


그림 7.13 정수 인덱스가 단어 임베딩으로 바뀌는 계산 과정

임베딩 레이어의 행과 각 단어 임베딩은 동일한 값을 가집니다. 사실 임베딩 레이어는 정수 인덱스에 저장된 단어의 수만큼 단어 임베딩을 가지고 있다가 필요할 때 꺼내쓸 수 있는 저장 공간과 같습니다.

그런데 자연어에는 미리 정해놓을 수 없을 정도로 많은 단어가 있기 때문에 보통은 정수 인덱스로 저장하지 않는 단어에 대한 임베딩 값을 별도로 마련해 놓습니다. 즉, 임베딩 레이어의 행 수가 10,000이라면 9,999는 미리 지정된 단어의 개수이고, 나머지 1은 지정되지 않은 단어를 위한 값입니다. 이것을 UNK(Unknown) 값이라고 합니다. 구현에 따라서는 공백을 의미하는 패딩(padding) 값을 위한 인덱스를 하나 더 확보해야 할 때도 있습니다. 이 경우에는 미리 지정된 단어를 9,998개만 써야 합니다.

<sup>11</sup> 《Deep Learning, NLP, and Representations》, 크리스토퍼 올라 블로그, <http://bit.ly/2M62MnZ>

임베딩 레이어에 대한 개념은 쉬운 편이지만 학습시키는 방법에는 여러 가지가 있습니다. 대표적인 방법은 Word2Vec<sup>12</sup>, GloVe<sup>13</sup>, FastText<sup>14</sup>, ELMo<sup>15</sup> 등이 있습니다. 이 방법론으로 미리 훈련된 임베딩 레이어의 가중치를 불러와서 사용하면 학습 시간을 절약할 수 있습니다. 하지만 이 책에서는 랜덤한 값에서 시작해서 가중치를 점점 적합한 값으로 학습시켜나가는 방법으로 임베딩 레이어를 사용하겠습니다.

이어지는 두 절에서는 지금까지 배운 주요 레이어들을 사용해 한글 자연어의 긍정, 부정 감성 분석과 자연어 생성이라는 두 가지 예제 코드를 살펴보겠습니다.

## 7.3 긍정, 부정 감성 분석

감성 분석(Sentiment Analysis)은 입력된 자연어 안의 주관적 의견, 감정 등을 찾아내는 문제입니다. 이 가운데 극성(polarity) 감성 분석은 문장의 긍정/부정이나 긍정/중립/부정을 분류합니다. 영화 리뷰나 음식점 리뷰는 데이터의 양이 많고 별점을 함께 달기 때문에 긍정/중립/부정 라벨링이 쉬워서 극성 감성 분석에 쉽게 적용할 수 있습니다.

이번 절에서는 네이버의 박은정 박사가 2015년에 발표한 《Naver sentiment movie corpus v1.0》<sup>16</sup>을 이용해 긍정/부정 감성 분석을 해보겠습니다. 여기에는 훈련 데이터로 15만 개, 테스트 데이터로 5만 개로 총 20만 개의 리뷰가 있습니다. 리뷰 중 10만 개는 별점이 1~4로 부정적인 리뷰이고 나머지 10만 개는 9~10으로 긍정적인 리뷰입니다. 별점 5~8에 해당하는 리뷰는 중립적이라고도 볼 수 있겠지만 이 데이터세트에서는 제외됐습니다.

먼저 데이터세트를 불러오겠습니다. 예제 7.19는 깃허브에 공개돼 있는 파일을 구글 코랩에 내려받는 코드입니다.

<sup>12</sup> T. Mikolov, I. Sutskever, K. Chen, G. Corrado, J. Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In Proceedings of NIPS 2013. <http://bit.ly/2KxeNQ1>

<sup>13</sup> <https://nlp.stanford.edu/projects/glove/>

<sup>14</sup> <https://fasttext.cc/>

<sup>15</sup> <https://aclweb.org/anthology/N18-1202>

<sup>16</sup> <https://github.com/e9t/nsmc/>

## [IN]

```
path_to_train_file = tf.keras.utils.get_file('train.txt', 'https://raw.githubusercontent.com/e9t/nsmc/master/ratings_train.txt')
path_to_test_file = tf.keras.utils.get_file('test.txt', 'https://raw.githubusercontent.com/e9t/nsmc/master/ratings_test.txt')
```

## [OUT]

```
Downloading data from https://raw.githubusercontent.com/e9t/nsmc/master/ratings_train.txt
14630912/14628807 [=====] - 0s 0us/step
Downloading data from https://raw.githubusercontent.com/e9t/nsmc/master/ratings_test.txt
4898816/4893335 [=====] - 0s 0us/step
```

다운로드가 완료되면 데이터를 메모리에 불러와야 합니다. 이때 데이터가 어떻게 생겼는지 간단하게 확인해볼 수 있습니다.

## [IN]

```
# 데이터를 메모리에 불러옵니다. 인코딩 형식으로 utf-8을 지정해야 합니다.
train_text = open(path_to_train_file, 'rb').read().decode(encoding='utf-8')
test_text = open(path_to_test_file, 'rb').read().decode(encoding='utf-8')

# 텍스트가 총 몇 자인지 확인합니다.
print('Length of text: {} characters'.format(len(train_text)))
print('Length of text: {} characters'.format(len(test_text)))
print()

# 처음 300자를 확인해봅니다.
print(train_text[:300])
```

## [OUT]

```
Length of text: 6937271 characters
Length of text: 2318260 characters

id      document label
9976970      아 더빙.. 진짜 짜증나네요 목소리      0
3819312      흠...포스터보고 초딩영화줄....오버연기조차 가볍지 않구나      1
```

10265843	너무재밌었다그래서보는것을추천한다	0
9045019	교도소 이야기구먼 ..솔직히 재미는 없다..평점 조정	0
6483659	사이몬페그의 익살스런 연기가 돋보였던 영화!스파이더맨에서 늙어보이기만 했던 커스틴 던스트가 너무나도 이뻐보였다	1
5403919	막 걸음마 땐 3세부터 초등학교 1학년생인 8살용영화.ㅋㅋㅋ...별반개도 아까움.	
0		
7797314	원작의	

데이터의 각 행은 탭 문자(\t)로 구분돼 있습니다. 처음의 id는 각 데이터의 고유한 id이고, document는 실제 리뷰 내용입니다. label은 긍정/부정을 나타내는 값으로, 0은 부정, 1은 긍정입니다.

이제 학습을 위한 훈련 데이터와 테스트 데이터를 만들어 보겠습니다. 입력(X)에 해당하는 자연어의 처리는 복잡한 과정이기 때문에 조금 뒤에서 다루기로 하고 일단 0, 1만 존재하는 출력(Y)부터 처리해 보겠습니다.

#### 예제 7.21 학습을 위한 정답 데이터(Y) 만들기

##### [IN]

```
train_Y = np.array([[int(row.split('\t')[2])] for row in train_text.split('\n')[1:] if
row.count('\t') > 0])
test_Y = np.array([[int(row.split('\t')[2])] for row in test_text.split('\n')[1:] if
row.count('\t') > 0])
print(train_Y.shape, test_Y.shape)
print(train_Y[:5])
```

##### [OUT]

```
(150000, 1) (50000, 1)
[[0]
 [1]
 [0]
 [0]
 [1]]
```

첫 번째 줄과 두 번째 줄은 먼저 각 텍스트를 개행 문자(\n)로 분리한 다음에 헤더에 해당하는 부분(id document label)을 제외한 나머지([1:])에 대해 각 행을 처리합니다. 각 행은 탭 문자(\t)로 나눠진 후에 2번째 원소(파이썬은 0부터 숫자를 세니까 실제로는 3번째 원소입니다)를 정수(integer)로 변환해서 저장합니다. 마지막에는 np.array로 결과 리스트를 감싸서 네트워크에 입력하기 쉽게 만듭니다.

훈련 데이터 Y의 첫 원소 다섯 개를 출력해보면 정답 라벨이 잘 들어있음을 확인할 수 있습니다.

그다음으로는 입력으로 쓸 자연어를 토큰화(Tokenization)하고 정제(Cleaning)해야 합니다. 토큰화란 자연어를 처리 가능한 작은 단위로 나누는 것으로, 여기서는 단어를 사용할 것이기 때문에 띄어쓰기 단위로 나누면 됩니다. 정제란 원하지 않는 입력이나 불필요한 기호 등을 제거하는 것입니다. 정제를 위한 함수로는 김윤 박사의 CNN\_sentence 깃허브 저장소<sup>17</sup>의 코드를 사용했습니다.

#### 예제 7.22 훈련 데이터의 입력(X) 정제

##### [IN]

```
import re
# From https://github.com/yoonkim/CNN_sentence/blob/master/process_data.py
def clean_str(string):
    string = re.sub(r"[^가-힣A-Za-z0-9(),!?\`]", " ", string)
    string = re.sub(r"\s", " \s", string)
    string = re.sub(r"\ve", " \ve", string)
    string = re.sub(r"\n\t", " \n\t", string)
    string = re.sub(r"\re", " \re", string)
    string = re.sub(r"\d", " \d", string)
    string = re.sub(r"\ll", " \ll", string)
    string = re.sub(r"\,", " , ", string)
    string = re.sub(r"\!", " ! ", string)
    string = re.sub(r"\(", " \( ", string)
    string = re.sub(r"\)", " \) ", string)
    string = re.sub(r"\?", " \? ", string)
    string = re.sub(r"\s{2,}", " ", string)
    string = re.sub(r"\{2,}", "\'', string)
    string = re.sub(r"\'", "", string)

    return string.lower()

train_text_X = [row.split('\t')[1] for row in train_text.split('\n')[1:] if row.count('\t') > 0]
train_text_X = [clean_str(sentence) for sentence in train_text_X]
# 문장을 띄어쓰기 단위로 단어 분리
sentences = [sentence.split(' ') for sentence in train_text_X]
for i in range(5):
    print(sentences[i])
```

<sup>17</sup> [https://github.com/yoonkim/CNN\\_sentence](https://github.com/yoonkim/CNN_sentence)

## [OUT]

```
['아', '더빙', '진짜', '짜증나네요', '목소리']
['흠', '포스터보고', '초딩영화줄', '오버연기조차', '가볍지', '않구나']
['너무재밌었다그래서보는것을추천한다']
['교도소', '이야기구먼', '솔직히', '재미는', '없다', '평점', '조정']
['사이몬페그의', '익살스런', '연기가', '돋보였던', '영화', '!', '스파이더맨에서',
'늙어보이기만', '했던', '커스틴', '던스트가', '너무나도', '이뻐보였다']
```

첫 줄의 `import re`는 정규표현식 라이브러리를 불러옵니다. `clean_str(string)` 함수는 다수의 정규표현식을 사용하고 있습니다만 첫 줄을 제외하면 세 번째 인수인 `string`에서 첫 번째 인수에 해당하는 내용을 찾아서 두 번째 인수로 단순히 교체해주는 것입니다.

```
string = re.sub(r"[^가-힣A-Za-z0-9(),!?\``]", " ", string)
```

첫 줄도 세 번째 인수에서 첫 번째 인수를 찾아서 두 번째 인수로 교체해주는 흐름은 같지만 특이한 점은 대괄호([])로 묶은 부분의 처음에 ^가 들어가 있다는 것입니다. 이 기호는 대괄호 안의 내용을 찾은 다음에, 그에 포함되지 않는 나머지 모두를 선택한다는 뜻입니다. 즉, 한글, 영문, 숫자, 괄호, 쉼표, 느낌표, 물음표, 작은따옴표('), 역따옴표(`)를 제외한 나머지는 모두 찾아서 공백(" ")으로 바꾸겠다는 뜻입니다.<sup>18</sup>

훈련 데이터의 처음 다섯 개를 출력해보면 구두점(.) 같은 기호가 삭제되고 단어 단위로 나눠진 데이터가 생긴 것을 확인할 수 있습니다.

그런데 네트워크에 입력하기 위한 데이터의 크기(문장의 길이)는 동일해야 하는데 현재는 각 문장의 길이가 다르기 때문에 문장의 길이를 맞춰야 합니다. 이를 위해서는 적당한 길이의 문장이 어느 정도인지 확인하고, 긴 문장은 줄이고 짧은 문장에는 공백을 의미하는 패딩(padding)을 채워넣어야 합니다. 각 문장의 길이를 그래프로 그려보겠습니다.

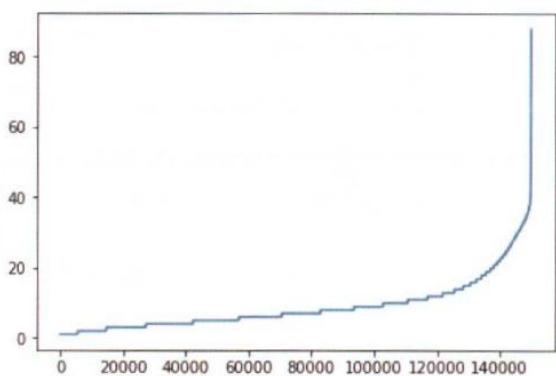
<sup>18</sup> 파이썬 정규표현식을 온라인에서 테스트해볼 수 있는 사이트로는 <https://regex101.com/> 있습니다. 이 사이트의 "REGULAR EXPRESSION"에 `[^가-힣A-Za-z0-9(),!?\``]`를 입력하면 정규표현식의 내용에 대한 해석이 "EXPLANATION"에 나옵니다.

## [IN]

```
import matplotlib.pyplot as plt
sentence_len = [len(sentence) for sentence in sentences]
sentence_len.sort()
plt.plot(sentence_len)
plt.show()

print(sum([int(l<=25) for l in sentence_len]))
```

## [OUT]



142587

그래프의 Y축은 문장의 단어 개수입니다. 15만 개의 문장 중에서 대부분이 40단어 이하로 구성돼 있음을 확인할 수 있습니다. 특히 25 단어 이하인 문장의 수는 142,587개로 전체의 95% 정도입니다. 따라서 기준이 되는 문장의 길이를 25 단어로 잡고 이 이상은 생략, 이 이하는 패딩으로 길이를 25로 맞춰주면 임베딩 레이어에 넣을 준비가 끝납니다.

또 하나 처리해야 하는 부분은 각 단어의 최대 길이를 조정하는 일입니다. 훈련 데이터의 5번째 문장에는 “스파이더맨에서”라는 단어가 있습니다. 이 단어와 “스파이더맨”, “스파이더맨이”, “스파이더맨을” 등 의 단어가 나온다고 가정할 때, 앞에서부터 5글자로 자르면 모두 “스파이더맨”이라는 한 단어가 됩니다. 이렇게 자르더라도 단어가 가진 의미가 어느 정도는 보존되기 때문에 적절한 길이로 자르면 여러 개의 단어에 분산될 수 있는 의미를 하나로 모을 수 있게 됩니다.<sup>19</sup> 예제 7.24는 위에서 설명한 단어 정제 및 문장 길이를 줄이는 작업을 합니다.

<sup>19</sup> 영어에서는 어간 추출(Stemming)과 표제어 추출(Lemmatization)이라는 기법으로 단어를 전처리합니다. 파이썬 라이브러리인 nltk를 사용할 때 어간 추출은 “lying”이라는 단어를 “ly”로 변환하고, 표제어 추출 함수 중 하나인 PorterStemmer()는 “lie”로 변환합니다. <https://www.nltk.org/book/ch03.html>

**[IN]**

```

sentences_new = []
for sentence in sentences:
    sentences_new.append([word[:5] for word in sentence][:25])
sentences = sentences_new
for i in range(5):
    print(sentences[i])

```

**[OUT]**

```

['아', '더빙', '진짜', '짜증나네요', '목소리']
['흠', '포스터보고', '초딩영화줄', '오버연기조', '가볍지', '않구나']
['너무재밌었']
['교도소', '이야기구먼', '솔직히', '재미는', '없다', '평점', '조정']
['사이몬페그', '익살스런', '연기가', '돋보였던', '영화', '!', '스파이더맨', '늙어보이기',
 '했던', '커스틴', '던스트가', '너무나도', '이뻐보였다']

```

“스파이더맨에서”가 “스파이더맨”으로 줄어든 것처럼 단어의 길이가 최대 다섯 글자로 줄어든 것을 확인 할 수 있습니다.

이제 앞에서 설명한 작업 중에서 짧은 문장을 같은 길이의 문장(25단어)으로 바꾸기 위한 패딩을 넣기 위해 `tf.keras`에서 제공하는 `pad_sequences`를 사용해 보겠습니다. 또 모든 단어를 사용하지 않고 출현 빈도가 가장 높은 일부 단어만 사용하기 위해 `Tokenizer`도 사용하겠습니다.

## 예제 7.25 Tokenizer와 pad\_sequences를 이용한 문장 전처리

**[IN]**

```

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

tokenizer = Tokenizer(num_words=20000)
tokenizer.fit_on_texts(sentences)
train_X = tokenizer.texts_to_sequences(sentences)
train_X = pad_sequences(train_X, padding='post')

print(train_X[:5])

```

## [OUT]

```
[[ 25 884 8 5795 1111 0 0 0 0 0 0 0 0  
  0 0 0 0 0 0 0 0 0 0 0 0 0 0  
  0]  
[ 588 5796 6697 0 0 0 0 0 0 0 0 0 0 0  
  0 0 0 0 0 0 0 0 0 0 0 0 0 0  
  0]  
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
  0 0 0 0 0 0 0 0 0 0 0 0 0 0  
  0]  
[ 71 346 31 35 10468 0 0 0 0 0 0 0 0  
  0 0 0 0 0 0 0 0 0 0 0 0 0 0  
  0]  
[ 106 5338 4 2 2169 869 573 0 0 0 0 0 0  
  0 0 0 0 0 0 0 0 0 0 0 0 0 0  
  0]]
```

먼저 import로 사용할 라이브러리들을 불러옵니다. 이 텍스트 전처리 라이브러리들은 케라스에서 원래 제공하던 것을 tf.keras에서도 동일하게 제공합니다. Tokenizer는 데이터에 출현하는 모든 단어의 개수를 세고 빈도 수로 정렬해서 num\_words에 지정된 만큼만 숫자로 반환하고, 나머지는 0으로 반환합니다. tokenizer.fit\_on\_texts(sentences)는 Tokenizer에 데이터를 실제로 입력합니다. 이 과정을 거친 뒤 tokenizer.texts\_to\_sequences(sentences)는 문장을 입력받아 숫자를 반환합니다. 마지막으로 pad\_sequences()는 입력된 데이터에 패딩을 더합니다.

출력에서 전처리된 문장을 확인할 수 있습니다. 첫 번째 문장에서 “아”는 25, “더빙”은 884 등의 숫자로 바뀌었습니다. 문장에서 사용하지 않는 부분은 0을 패딩으로 넣어서 입력의 길이인 25를 맞춰줍니다. pad\_sequences()의 padding 인수에는 2가지 옵션이 있는데, 이 가운데 'pre'는 문장의 앞에 패딩을 넣고, 'post'는 문장의 뒤에 패딩을 넣습니다. 여기서는 post를 사용했습니다.

출력의 세 번째 문장은 모두 0으로 표시되고 있는데, 이는 “너무재밌었”이라는 단어가 빈도 수에서 상위 20,000개에 들지 못해 패딩과 같은 0으로 처리된 것입니다. 다섯 번째 문장도 단어 수에 비해 실제로 0이 아닌 값을 가진 데이터의 길이가 7로 짧은 편인데 여기에도 Tokenizer에서 걸러진 단어가 많다는 것을 확인할 수 있습니다.

말로만 설명하면 Tokenizer가 어떻게 동작하는지 감이 잡히지 않을 수도 있을 것 같아서 간단한 코드로 이해를 돋겠습니다. Tokenizer에서 사용할 단어의 최대 숫자는 20,000으로 지정돼 있기 때문에 0번부터 19,999번 단어까지는 해당 인덱스의 숫자를 반환하겠지만 20,000번부터는 패딩 값과 같은 0을 반환할 것입니다. Tokenizer의 인덱스와 단어는 Tokenizer.index\_word에 저장돼 있습니다.

#### 예제 7.26 Tokenizer의 동작 확인

##### [IN]

```
print(tokenizer.index_word[19999])
print(tokenizer.index_word[20000])
temp = tokenizer.texts_to_sequences(['$$$$', '경우는', '잊혀질', '연기가'])
print(temp)
temp = pad_sequences(temp, padding='post')
print(temp)
```

##### [OUT]

```
경우는
잊혀질
[[[], [19999], [], [106]]]
[[    0]
 [19999]
 [    0]
 [ 106]]
```

예제 7.26에서는 Tokenizer.index\_word에 저장돼 있는 19,999번째 단어와 20,000번째 단어를 확인해본 뒤, 이 단어들로 구성된 문장을 Tokenizer에 넣어봤습니다. 또 첫 번째 단어는 리뷰에서 잘 쓰이지 않았을 것 같은 “\$\$\$\$”입니다. 네 번째 단어는 “연기가”라는 단어인데 훈련 데이터의 다섯 번째 문장의 일부입니다.

texts\_to\_sequences()에서 반환하는 데이터는 19,999번째 단어인 “경우는”과 “연기가”만 의미 있는 데이터로 남기고 나머지는 공백으로 반환합니다. 이 공백은 pad\_sequences()를 통과하면 0으로 바뀝니다. 여기서는 pad\_sequences()의 maxlen 인수가 지정되지 않았기 때문에 입력된 문장 전체의 길이 중 가장 긴 길이로 문장의 길이를 맞춥니다. 입력 문장이 하나이기 때문에 입력과 출력 문장의 길이는 동일합니다.

이제 실제로 네트워크를 정의하고 학습시켜보겠습니다. 먼저 임베딩 레이어와 LSTM 레이어를 연결한 뒤 마지막에 Dense 레이어의 소프트맥스 활성화함수를 사용해 긍정/부정을 분류하는 네트워크를 정의해 보겠습니다.

## [IN]

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(20000, 300, input_length=25),
    tf.keras.layers.LSTM(units=50),
    tf.keras.layers.Dense(2, activation='softmax')
])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

## [OUT]

```
Model: "sequential_31"

Layer (type)          Output Shape         Param #
embedding_19 (Embedding)    (None, 25, 300)       6000000
unified_lstm_34 (UnifiedLSTM (None, 50)        70200
dense_42 (Dense)           (None, 2)            102

Total params: 6,070,302
Trainable params: 6,070,302
Non-trainable params: 0
```

위의 네트워크에서 임베딩 레이어는 시퀀셜 모델의 첫 번째 레이어이기 때문에 입력 형태에 대한 정의가 필요합니다. `input_length` 인수를 25로 지정해서 각 문장에 들어있는 25개의 단어를 길이 300의 임베딩 벡터로 변환합니다.

네트워크의 `loss`는 `sparse_categorical_crossentropy`를 사용했습니다. 여러 개의 정답 중 하나를 맞추는 분류 문제일 때 `categorical_crossentropy`를 사용하고, `sparse`는 정답인 Y가 희소 행렬일 때 사용한다고 5장과 6장에서 설명했습니다.

그럼 이제 네트워크를 학습시켜보겠습니다.

**[IN]**

```
history = model.fit(train_X, train_Y, epochs=5, batch_size=128, validation_split=0.2)
```

**[OUT]**

```
Train on 120000 samples, validate on 30000 samples
Epoch 1/5
120000/120000 [=====] - 75s 623us/sample - loss: 0.4351 - accuracy: 0.7837 - val_loss: 0.3907 - val_accuracy: 0.8186
Epoch 2/5
120000/120000 [=====] - 69s 578us/sample - loss: 0.3241 - accuracy: 0.8485 - val_loss: 0.3912 - val_accuracy: 0.8168
Epoch 3/5
120000/120000 [=====] - 69s 578us/sample - loss: 0.2721 - accuracy: 0.8683 - val_loss: 0.4313 - val_accuracy: 0.8180
Epoch 4/5
120000/120000 [=====] - 68s 571us/sample - loss: 0.2297 - accuracy: 0.8871 - val_loss: 0.5035 - val_accuracy: 0.8091
Epoch 5/5
120000/120000 [=====] - 69s 575us/sample - loss: 0.1983 - accuracy: 0.9017 - val_loss: 0.5381 - val_accuracy: 0.8062
```

데이터가 많기 때문에 한번에 학습하는 데이터의 양인 batch\_size를 128로 설정했고, 5에포크만 학습을 시켰습니다. 학습 과정에서 loss는 꾸준히 감소하지만 val\_loss는 점점 증가하는 것을 확인할 수 있습니다. 이는 네트워크가 과적합되고 있다는 의미입니다. 학습 결과를 그래프로 확인해보겠습니다.

**[IN]**

```
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], 'b-', label='loss')
plt.plot(history.history['val_loss'], 'r--', label='val_loss')
plt.xlabel('Epoch')
plt.legend()
```

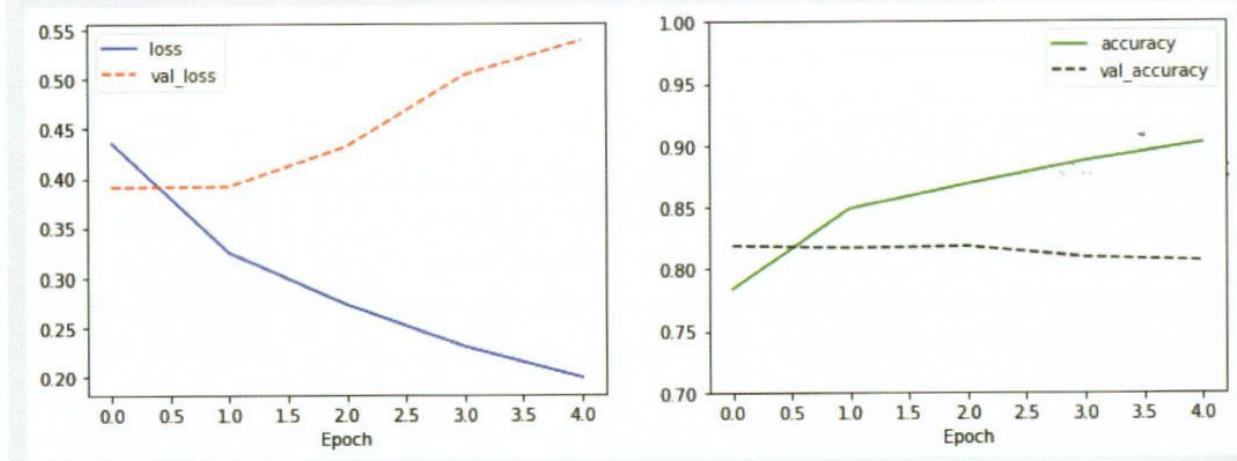
```

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], 'g-', label='accuracy')
plt.plot(history.history['val_accuracy'], 'k--', label='val_accuracy')
plt.xlabel('Epoch')
plt.ylim(0.7, 1)
plt.legend()

plt.show()

```

[OUT]



왼쪽 그래프에서 붉은색 점선으로 표시된 val\_loss의 증가 추세가 눈에 띕니다. 오른쪽 그래프에서 val\_accuracy도 점점 떨어지고 있는 것으로 봐서 네트워크가 과적합되는 것으로 보입니다. 과적합의 이유는 임베딩 레이어를 랜덤한 값에서부터 시작해서 학습시키기 때문에 각 단어를 나타내는 벡터의 품질이 좋지 않아서입니다. 이를 개선하기 위한 방법으로 임베딩 레이어를 별도로 학습시켜서 네트워크에 불러와서 사용하거나 RNN이 아닌 CNN을 사용하는 방법이 있습니다.

학습된 네트워크가 테스트 데이터는 어떻게 평가할까요? 확인을 위해 test\_text에도 train\_text와 같은 변환 과정을 거쳐서 test\_X를 만들고 model.evaluate()로 평가해 보겠습니다. 이때 한 가지 주목해야 할 것은 train\_X를 만들 때 학습시켰던 Tokenizer를 어떤 변경 없이 그대로 사용하고 있다는 것입니다. 훈련 데이터와는 다르게 테스트 데이터에서는 어떤 단어가 나타날지 모르기 때문에 Tokenizer는 훈련 데이터로만 학습시켜야 합니다. 이는 4.4절에서 설명한 데이터 정규화와 같은 이유입니다. 테스트 데이터는 우리의 손에 없다고 생각하고 작업해야 합니다.

**[IN]**

```

test_text_X = [row.split('\t')[1] for row in test_text.split('\n')[1:] if row.count('\t') > 0]
test_text_X = [clean_str(sentence) for sentence in test_text_X]
sentences = [sentence.split(' ') for sentence in test_text_X]
sentences_new = []
for sentence in sentences:
    sentences_new.append([word[:5] for word in sentence][:25])
sentences = sentences_new

test_X = tokenizer.texts_to_sequences(sentences)
test_X = pad_sequences(test_X, padding='post')

model.evaluate(test_X, test_Y, verbose=0)

```

**[OUT]**

```
[0.547995473408699, 0.80038]
```

테스트 데이터의 정확도는 80%로 나왔습니다. 이는 검증 데이터의 정확도와 비슷한 값입니다. 그렇다면 임의의 문장에 대한 감성 분석은 어떨까요? 순환 신경망이 입력의 변화에 따라 값이 변한다는 것을 확인하기 위해 하나의 문장을 잘라서 앞에서부터 차례로 입력해보겠습니다.

## 예제 7.31 임의의 문장에 대한 감성 분석 결과 확인

**[IN]**

```

test_sentence = '재미있을 줄 알았는데 완전 실망했다. 너무 졸리고 돈이 아까웠다.'
test_sentence = test_sentence.split(' ')
test_sentences = []
now_sentence = []
for word in test_sentence:
    now_sentence.append(word)
    test_sentences.append(now_sentence[:])

test_X_1 = tokenizer.texts_to_sequences(test_sentences)
test_X_1 = pad_sequences(test_X_1, padding='post', maxlen=25)
prediction = model.predict(test_X_1)
for idx, sentence in enumerate(test_sentences):
    print(sentence)
    print(prediction[idx])

```

## [OUT]

```
[ '재미있을']
[ 0.41499388 0.5850061 ]
[ '재미있을', '줄']
[ 0.46383956 0.53616047]
[ '재미있을', '줄', '알았는데']
[ 0.5084595 0.4915405]
[ '재미있을', '줄', '알았는데', '완전']
[ 0.52025294 0.4797471 ]
[ '재미있을', '줄', '알았는데', '완전', '실망했다.']
[ 0.52025294 0.4797471 ]
[ '재미있을', '줄', '알았는데', '완전', '실망했다.', '너무']
[ 0.59737766 0.40262237]
[ '재미있을', '줄', '알았는데', '완전', '실망했다.', '너무', '줄리고']
[ 0.99524826 0.00475175]
[ '재미있을', '줄', '알았는데', '완전', '실망했다.', '너무', '줄리고', '돈이']
[ 0.9979342 0.00206574]
[ '재미있을', '줄', '알았는데', '완전', '실망했다.', '너무', '줄리고', '돈이', '아까웠다.']
[ 0.9979342 0.00206574]
```

출력은 문장의 변화에 따른 감성 분석 예측 결과입니다. 처음에 “재미있을”이라는 단어만 입력됐을 때는 긍정의 확률이 58.5%로 부정보다 높습니다. 그런데 이 확률은 다른 단어들이 입력되며 꾸준히 줄어들고, 특히 “너무” 다음에 “줄리고”가 나왔을 때 99%의 확률로 부정적 감성을 예측합니다. test\_sentence에 들어가는 입력 문장을 바꿔가면서 테스트를 해보면 재미있는 결과들을 확인할 수 있습니다.<sup>20</sup>

## 7.4 자연어 생성

현재 테슬라(Tesla)의 AI Director로 있는 안드레아 카르파티(Andrej Karpathy)는 스탠퍼드 박사과정일 때 개인 블로그<sup>21</sup>에 딥러닝에 대한 여러 개의 글을 올렸는데, 그중 “The Unreasonable Effectiveness of Recurrent Neural Networks”<sup>22</sup>라는 글은 문자 단위의 순환 신경망이 놀라운 결과

<sup>20</sup> 이 예제의 인터랙티브 버전은 Keras.js 홈페이지의 Bidirectional LSTM 예제에서 확인할 수 있습니다(<http://bit.ly/2Z4vwnj>). LOAD SAMPLE TEXT 버튼을 누르면 미리 저 장된 문장을 불러오고 긍정/부정을 평가합니다. 또 입력을 변화시키면 그에 맞게 긍정/부정 출력이 변화합니다. Bidirectional LSTM은 LSTM을 양방향으로 연결한 것으로, 케리스에서는 LSTM 등의 순환 신경망을 다른 레이어로 감싸서 쉽게 구현할 수 있습니다.

<sup>21</sup> <http://karpathy.github.io/>

<sup>22</sup> <http://bit.ly/2GVIE3C>

물을 만들어낼 수 있다는 사실을 조명해서 주목받았습니다. 이 글에서는 셰익스피어의 희곡, 소스코드, Latex 등을 문자 단위의 순환 신경망으로 생성해서 비슷한 구조를 가진 문서를 재생산하는 데 순환 신경망이 효과적이라는 것을 보여줬습니다.

*Proof.* Omitted. □

**Lemma 0.1.** Let  $\mathcal{C}$  be a set of the construction.

Let  $\mathcal{C}$  be a gerber covering. Let  $\mathcal{F}$  be a quasi-coherent sheaves of  $\mathcal{O}$ -modules. We have to show that

$$\mathcal{O}_{\mathcal{O}_X} = \mathcal{O}_X(\mathcal{L})$$

*Proof.* This is an algebraic space with the composition of sheaves  $\mathcal{F}$  on  $X_{\text{state}}$ , we have

$$\mathcal{O}_X(\mathcal{F}) = \{\text{morph}_1 \times_{\mathcal{O}_X} (\mathcal{G}, \mathcal{F})\}$$

where  $\mathcal{G}$  defines an isomorphism  $\mathcal{F} \rightarrow \mathcal{F}$  of  $\mathcal{O}$ -modules. □

**Lemma 0.2.** This is an integer  $Z$  is injective.

*Proof.* See Spaces, Lemma ??.

**Lemma 0.3.** Let  $S$  be a scheme. Let  $X$  be a scheme and  $X$  is an affine open covering. Let  $\mathcal{U} \subset X$  be a canonical and locally of finite type. Let  $X$  be a scheme. Let  $X$  be a scheme which is equal to the formal complex.

The following to the construction of the lemma follows.

Let  $X$  be a scheme. Let  $X$  be a scheme covering. Let

$$b : X \rightarrow Y' \rightarrow Y \rightarrow Y \rightarrow Y' \times_X Y \rightarrow X.$$

be a morphism of algebraic spaces over  $S$  and  $Y$ .

*Proof.* Let  $X$  be a nonzero scheme of  $X$ . Let  $X$  be an algebraic space. Let  $\mathcal{F}$  be a quasi-coherent sheaf of  $\mathcal{O}_X$ -modules. The following are equivalent

- (1)  $\mathcal{F}$  is an algebraic space over  $S$ .
- (2) If  $X$  is an affine open covering.

Consider a common structure on  $X$  and  $X$  the functor  $\mathcal{O}_X(U)$  which is locally of finite type. □

This since  $\mathcal{F} \in \mathcal{F}$  and  $x \in \mathcal{G}$  the diagram

```

    \begin{CD}
    S @>>> G \\
    @V VV @VV V \\
    \text{Spec}(K_G) @>>> O_X \\
    @V VV @VV V \\
    X @= X
    \end{CD}
  
```

is a limit. Then  $\mathcal{G}$  is a finite type and assume  $S$  is a flat and  $\mathcal{F}$  and  $\mathcal{G}$  is a finite type  $f_*$ . This is of finite type diagrams, and

- the composition of  $\mathcal{G}$  is a regular sequence.
- $\mathcal{O}_X$  is a sheaf of rings. □

*Proof.* We have see that  $X = \text{Spec}(R)$  and  $\mathcal{F}$  is a finite type representable by algebraic space. The property  $\mathcal{F}$  is a finite morphism of algebraic stacks. Then the cohomology of  $X$  is an open neighbourhood of  $U$ . □

*Proof.* This is clear that  $\mathcal{G}$  is a finite presentation, see Lemmas ??.

A reduced above we conclude that  $U$  is an open covering of  $\mathcal{C}$ . The functor  $\mathcal{F}$  is a  $\text{-field}$

$$\mathcal{O}_{X,x} \rightarrow \mathcal{F}_{x,-1}(\mathcal{O}_{X_{\text{state}}}) \rightarrow \mathcal{O}_{X,x}^{-1}\mathcal{O}_{X,x}(\mathcal{O}_{X,x}^n)$$

is an isomorphism of covering of  $\mathcal{O}_{X,x}$ . If  $\mathcal{F}$  is the unique element of  $\mathcal{F}$  such that  $X$  is an isomorphism.

The property  $\mathcal{F}$  is a disjoint union of Proposition ?? and we can filtered set of presentations of a  $\mathcal{O}_X$ -algebra with  $\mathcal{F}$  are opens of finite type over  $S$ . □

If  $\mathcal{F}$  is a scheme theoretic image points. □

If  $\mathcal{F}$  is a finite direct sum  $\mathcal{O}_{X,x}$  is a closed immersion, see Lemma ??.

This is a sequence of  $\mathcal{F}$  is a similar morphism.

그림 7.14 안드레아 카르파티가 생성한 Latex를 컴파일한 결과. 의미는 해석하기 힘들지만 생김새는 그럴듯합니다.

이번 절에서는 한글 원본 텍스트를 자소 단위와 단어 단위로 나눠서 순환 신경망으로 생성해 보겠습니다.<sup>23</sup>

## 7.4.1 단어 단위 생성

7.3절에서 단어 단위로 입력 문장을 분리한 뒤에 감성 분석을 해봤기 때문에 익숙한 코드로 다시 한번 작업해보겠습니다. 이번에 작업할 원본 텍스트는 국사편찬위원회에서 제공하는 조선왕조실록 국문 번역본입니다. 먼저 데이터를 다운로드합니다.<sup>24</sup>

<sup>23</sup> 이 부분의 코드는 다음 두 곳에서 일정 부분을 참조했습니다.

- 《순환 신경망을 활용한 문자열 생성》, 텐서플로 홈페이지: <http://bit.ly/2TIE7tZ>
- 《Using LSTMs, see if you can write Shakespeare!》, deeplearning.ai: <http://bit.ly/2yQOUHf>

<sup>24</sup> 조선왕조실록 전체 텍스트 파일의 크기는 약 400MB 정도지만 여기서는 일부분의 일부(6MB)만 잘라낸 파일을 사용하겠습니다. 이 정도로도 텍스트를 생성하는 데는 충분합니다.

## [IN]

```
path_to_file = tf.keras.utils.get_file('input.txt', 'http://bit.ly/2Mc3SOV')
```

## [OUT]

```
Downloading data from http://bit.ly/2Mc3SOV
62013440/62012502 [=====] - 1s 0us/step
```

데이터 파일은 약 62MB 정도로 지금까지 사용해온 데이터들에 비해 크기가 큰 편입니다. 이 데이터를 7.3절에서 작업한 것처럼 메모리에 불러온 다음 글자 수와 데이터의 첫 부분을 확인해봅니다.

## [IN]

```
# 데이터를 메모리에 불러옵니다. 인코딩 형식으로 utf-8을 지정해야 합니다.
train_text = open(path_to_file, 'rb').read().decode(encoding='utf-8')

# 텍스트가 총 몇 자인지 확인합니다.
print('Length of text: {} characters'.format(len(train_text)))
print()

# 처음 100자를 확인해봅니다.
print(train_text[:100])
```

## [OUT]

```
Length of text: 26265493 characters
```

태조 이성계 선대의 가계. 목조 이안사가 전주에서 삼척·의주를 거쳐 알동에 정착하다  
태조 강현 지인 계운 성문 신무 대왕(太祖康獻至仁啓運聖文神武大王)의 성은 이씨(李氏)요, 휘

조선왕조실록답게 한자가 많은 부분을 차지하는 것을 볼 수 있습니다. 여기서는 단어 기반의 생성을 하려고 하기 때문에 한자와 한자가 들어간 팔호는 생략하겠습니다.

## [IN]

```

import re
# From https://github.com/yoonkim/CNN_sentence/blob/master/process_data.py
def clean_str(string):
    string = re.sub(r"[^가-힣A-Za-z0-9(),!?\`]", " ", string)
    string = re.sub(r"\ll", " ll", string)
    string = re.sub(r",", " , ", string)
    string = re.sub(r"!", " ! ", string)
    string = re.sub(r"\(", " ", string)
    string = re.sub(r"\)", " ", string)
    string = re.sub(r"\?", " \? ", string)
    string = re.sub(r"\s{2,}", " ", string)
    string = re.sub(r"\'{2,}", "\'", string)
    string = re.sub(r"\\"", " ", string)

    return string

train_text = train_text.split('\n')
train_text = [clean_str(sentence) for sentence in train_text]
train_text_X = []
for sentence in train_text:
    train_text_X.extend(sentence.split(' '))
    train_text_X.append('\n')

train_text_X = [word for word in train_text_X if word != '']

print(train_text_X[:20])

```

## [OUT]

```

['태조', '이성계', '선대의', '가계', '목조', '이안사가', '전주에서', '삼척', '의주를',
'거쳐', '알동에', '정착하다', '\n', '태조', '강현', '지인', '계운', '성문', '신무', '대왕']

```

조선왕조실록에는 영문 텍스트가 없기 때문에 `clean_str()` 함수에서 영문 관련 처리는 삭제했습니다. 그리고 한자와 함께 들어가 있는 괄호도 삭제했습니다. 또 개행 문자(\n)의 보존을 위해 텍스트를 먼저 개행 문자로 나눈 뒤에 다시 합칠 때 개행 문자를 추가했습니다. 정제 과정을 마친 출력에는 한자어와 괄호 등이 보이지 않습니다.

다음으로 할 작업은 단어를 토큰화하는 것입니다. 그런데 여기서는 7.3절에서 썼던 Tokenizer를 쓰지 않고 직접 토큰화하겠습니다. 단어의 수가 너무 많기도 하고, 모든 단어를 사용할 것이기 때문에 단어의 빈도 수로 단어를 정렬하는 Tokenizer를 쓰면 불필요하게 많은 계산 시간을 쓰게 됩니다.

#### 예제 7.35 단어 토큰화

##### [IN]

```
# 단어의 set을 만듭니다.  
vocab = sorted(set(train_text_X))  
vocab.append('UNK')  
print ('{} unique words'.format(len(vocab)))  
  
# vocab list를 숫자로 매핑하고, 반대도 실행합니다.  
word2idx = {u:i for i, u in enumerate(vocab)}  
idx2word = np.array(vocab)  
  
text_as_int = np.array([word2idx[c] for c in train_text_X])  
  
# word2idx 의 일부를 알아보기 쉽게 출력해 봅니다.  
print('{}')  
for word,_ in zip(word2idx, range(10)):  
    print(' {:4s}: {}'.format(repr(word), word2idx[word]))  
print(' ...\\n')  
  
print('index of UNK: {}'.format(word2idx['UNK']))
```

##### [OUT]

```
332640 unique words
```

```
{  
    '\n': 0,  
    '!': 1,  
    ',': 2,  
    '000명으로': 3,  
    '001': 4,  
    '002': 5,  
    '003': 6,  
    '004': 7,  
    '005': 8,  
    '006': 9,  
    ...
```

```
}
```

```
index of UNK: 332639
```

2번째 줄에서 텍스트에 들어간 각 단어가 중복되지 않는 리스트를 만든 다음에, 3번째 줄에서는 텍스트에 존재하지 않는 토큰을 나타내는 'UNK'를 넣습니다. 총 단어 수는 332,640이고, 이 중 'UNK'의 인덱스는 332,639입니다. 이 인덱스는 나중에 임의의 문장을 입력할 때 텍스트에 미리 준비돼 있지 않았던 단어를 쓸 수 있기 때문에 그에 대한 토큰으로 쓰일 것입니다.

토큰화가 잘 됐는지 확인해보기 위해 처음 20개의 단어에 대한 토큰을 출력해보겠습니다.

#### 예제 7.36 토큰 데이터 확인

##### [IN]

```
print(train_text_X[:20])  
print(text_as_int[:20])
```

##### [OUT]

```
['태조', '이성계', '선대의', '가계', '목조', '이안사가', '전주에서', '삼척', '의주를',  
'거쳐', '알동에', '정착하다', '\n', '태조', '강현', '지인', '계운', '성문', '신무', '대왕']  
[299305 229634 161443 17430 111029 230292 251081 155087 225462 29027  
190295 256129 0 299305 25624 273553 36147 163996 180466 84413]
```

“태조”는 299,305로, “이성계”는 229,634로, 개행 문자는 0으로 토큰 변환이 된 것을 확인할 수 있습니다.

이제 학습을 위한 데이터세트를 만들어보겠습니다. 여기서 기존의 `train_X`, `train_Y`를 넘파이 array로 만드는 방식이 아닌 `tf.data.Dataset`를 이용해보겠습니다. `Dataset`의 장점은 간단한 코드로 데이터 섞기, 배치(batch) 수만큼 자르기, 다른 `Dataset`에 매핑하기 등을 빠르게 수행할 수 있다는 것입니다.

#### 예제 7.37 기본 데이터세트 만들기

##### [IN]

```
seq_length = 25  
examples_per_epoch = len(text_as_int) // seq_length  
sentence_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)
```

```
sentence_dataset = sentence_dataset.batch(seq_length+1, drop_remainder=True)
for item in sentences_dataset.take(1):
    print(idx2word[item.numpy()])
    print(item.numpy())
```

#### [OUT]

```
[299305 229634 161443 17430 111029 230292 251081 155087 225462 29027
 190295 256129      0 299305 25624 273553 36147 163996 180466 84413
 224182 164549 230248 210912      2 330313]
['태조' '이성계' '선대의' '가계' '목조' '이안사가' '전주에서' '삼척' '의주를' '거쳐' '알동에'
'정착하다'
'\n' '태조' '강현' '지인' '계운' '성문' '신무' '대왕' '의' '성은' '이씨' '요' ',,' '휘']
```

여기서는 seq\_length를 25로 설정해서 25개의 단어가 주어졌을 때 다음 단어를 예측하도록 데이터를 만들려고 합니다.

```
sentence_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)
```

Dataset를 생성하는 코드는 위와 같이 한 줄로 간단합니다.

```
sentence_dataset = sentence_dataset.batch(seq_length+1, drop_remainder=True)
```

Dataset에 쓰이는 batch() 함수는 Dataset에서 한번에 반환하는 데이터의 숫자를 지정합니다. 여기서는 seq\_length+1을 지정했는데, 처음의 25개 단어와 그 뒤에 오는 정답이 될 1단어를 합쳐서 함께 반환하기 위해서입니다. 또 drop\_remainder=True 옵션으로 남는 부분은 버리도록 했습니다. 출력에서 의도한 대로 26단어가 반환되는 것을 확인할 수 있습니다.

이제 이렇게 만들어진 Dataset로 새로운 Dataset를 만들어 보겠습니다. 26개의 단어가 각각 입력과 정답으로 묶어서 ([25단어], 1단어) 형태의 데이터를 반환하게 만드는 Dataset를 만드는 것입니다. 코드는 다음과 같습니다.

## [IN]

```
def split_input_target(chunk):
    return [chunk[:-1], chunk[-1]]

train_dataset = sentence_dataset.map(split_input_target)
for x,y in train_dataset.take(1):
    print(idx2word[x.numpy()])
    print(x.numpy())
    print(idx2word[y.numpy()])
    print(y.numpy())
```

## [OUT]

```
['태조' '이성계' '선대의' '가계' '목조' '이안사가' '전주에서' '삼척' '의주를' '거쳐' '알동에'
'정착하다'
'\n' '태조' '강현' '지인' '계운' '성문' '신무' '대왕' '의' '성은' '이씨' '요' ','
[299305 229634 161443 17430 111029 230292 251081 155087 225462 29027
190295 256129      0 299305 25624 273553 36147 163996 180466 84413
224182 164549 230248 210912      2]
회
330313
```

먼저 26개의 단어를 25단어, 1단어로 잘라주는 함수를 `split_input_target(chunk)`로 정의한 다음 이 함수를 기존 `sentence_dataset`에 `map()` 함수<sup>25</sup>를 이용해 새로운 Dataset인 `train_dataset`을 만듭니다.

그 다음으로는 Dataset의 데이터를 섞고, batch size를 다시 설정합니다.

## 예제 7.39 데이터세트 shuffle, batch 설정

```
BATCH_SIZE = 128
steps_per_epoch = examples_per_epoch // BATCH_SIZE
BUFFER_SIZE = 10000

train_dataset = train_dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)
```

<sup>25</sup> 파이썬에서 기본으로 제공되는 `map`과는 다른, Dataset에 정의돼 있는 `map` 함수입니다. 텐서플로 홈페이지의 `map` 함수 설명 부분 참고(<http://bit.ly/2GZTyW1>)

빠른 학습을 위해 한번에 128개의 데이터를 학습하게 했고, 데이터를 섞을 때의 BUFFER\_SIZE는 10,000으로 설정했습니다. tf.data는 이론적으로 무한한 데이터에 대해 대응 가능하기 때문에 한번에 모든 데이터를 섞지 않습니다. 따라서 버퍼에 일정한 양의 데이터를 올려놓고 섞는데, 그 사이즈를 10,000으로 설정한 것입니다.

이제 생성 모델을 정의해 보겠습니다.

#### 예제 7.40 단어 단위 생성 모델 정의

##### [IN]

```
total_words = len(vocab)
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(total_words, 100, input_length=seq_length),
    tf.keras.layers.LSTM(units=100, return_sequences=True),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.LSTM(units=100),
    tf.keras.layers.Dense(total_words, activation='softmax')
])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

##### [OUT]

Model: "sequential\_4"

Layer (type)	Output Shape	Param #
embedding_5 (Embedding)	(None, 25, 100)	33264000
unified_lstm_10 (UnifiedLSTM)	(None, 25, 100)	80400
dropout_5 (Dropout)	(None, 25, 100)	0
unified_lstm_11 (UnifiedLSTM)	(None, 100)	80400
dense_7 (Dense)	(None, 332640)	33596640

Total params: 67,021,440

Trainable params: 67,021,440

Non-trainable params: 0

생성 모델의 초반부는 7.3절과 비슷하게 임베딩 레이어와 LSTM 레이어로 구성돼 있습니다. LSTM 레이어는 2층으로 쌓여 있고 중간에 과적합을 막기 위한 드롭아웃 레이어를 배치했습니다. 마지막에는 Dense 레이어를 배치해서 소프트맥스 활성화함수로 주어진 입력에 대해 332,640개의 단어 중 어떤 단어를 선택해야 하는지를 고르게 됩니다.

그럼 이제 학습을 시켜보겠습니다.

#### 예제 7.41 단어 단위 생성 모델 학습

[IN]

```
from tensorflow.keras.preprocessing.sequence import pad_sequences

def testmodel(epoch, logs):
    if epoch % 5 != 0 and epoch != 49:
        return
    test_sentence = train_text[0]

    next_words = 100
    for _ in range(next_words):
        test_text_X = test_sentence.split(' ')[-seq_length:]
        test_text_X = np.array([word2idx[c] if c in word2idx else word2idx['UNK'] for c in test_text_X])
        test_text_X = pad_sequences([test_text_X], maxlen=seq_length, padding='pre',
                                   value=word2idx['UNK'])

        output_idx = model.predict_classes(test_text_X)
        test_sentence += ' ' + idx2word[output_idx[0]]

    :
    print()
    print(test_sentence)
    print()

testmodelcb = tf.keras.callbacks.LambdaCallback(on_epoch_end=testmodel)

history = model.fit(train_dataset.repeat(), epochs=50, steps_per_epoch=steps_per_epoch,
callbacks=[testmodelcb], verbose=2)
```

## [OUT]

Epoch 1/50

태조 이성계 선대의 가계 목조 이안사가 전주에서 삼척 의주를 거쳐 알동에 정착하다 , , , , ,

533/533 - 255s - loss: 9.3702 - accuracy: 0.0723

(생략)

Epoch 11/50

태조 이성계 선대의 가계 목조 이안사가 전주에서 삼척 의주를 거쳐 알동에 정착하다 등 것입니다  
하니 , 임금이 말하기를 ,  
상참을 받고 , 임금이 아뢰기를 ,  
이 받고 그 일을 다 다 가지고 주고 , 그 거느리고 다 화살을 내어 주고 , 이는 아뢰기를 ,  
함길도 의 서쪽에 서향하여 없는 것입니다 또 아뢰기를 ,  
이 받고 반드시 보내어 가서 가지고 가서 주고 , 그 거느리고 의하여 서향하여 있으니 , 그 받고 ,  
아뢰기를 ,  
그 일을 가지고 서향하여 서게 하고 , 그 받고 몸을 아뢰기를 ,  
예조에서 아뢰기를 ,  
함길도 의 서쪽에 인도하여 가지고 주고 , 아뢰기를 ,  
평안도 의 서쪽에 서향하여

533/533 - 250s - loss: 6.0086 - accuracy: 0.1822

(생략)

Epoch 31/50

태조 이성계 선대의 가계 목조 이안사가 전주에서 삼척 의주를 거쳐 알동에 정착하다 되고 , 청컨대  
본조 수는 없는데 , 무릇 여러 일은 이미 심히 알지 자는 서로 알 것이다 또한 이미 여러 사람이 심히  
말을 사람을 얻게 합니다 이에 갖추어 아뢰니 , 드디어 세자에게 거동하여 사실을 묻게 하였으나 , 임금이  
말하기를 ,  
이 사람을 거느리고 가서 와서 이르기를 ,  
이 사람을 보내 토산물을 바쳤다 나도 아뢴 않겠는가  
하였다 임금이 말하기를 ,  
전하께서 전지 에서 함께 사사로이 듣고 아뢰게 하고 , 만일 잘못 묻지는 를 올려 국문 하여 계문  
하여 이러하였다  
대간 으로 사제 를 하사하다  
대간 에 내리니 , 전하께서 다시 와서 가서 잘 알지 것은

533/533 - 250s - loss: 1.9361 - accuracy: 0.6555

(생략)

Epoch 50/50

태조 이성계 선대의 가계 목조 이안사가 전주에서 삼척 의주를 거쳐 알동에 정착하다 들을 것과 , 본조  
가 없으면 서향하게 것입니다 만약 먼저 단 가 있고 , 한 법이 같이 같이 되어 한 일이 있을 일이  
없습니다 그러나 , 이제부터는 진휼사 에 술을 받들어 그 자리에 매우 옮지 를 나아가서 , 내가 그 아비를  
가서 온 사람은 한번 가상히 부임 하여 이러한 때 아울러 국문 하여 외방 전의 일수 를 받들어 잡회 를  
나아가서 나아가서 나가고 , 이어서 병법 과 악 을 외유 일어나고 , 의장 을 더하여 적당히 판위 에 올라  
그치게 하되 , 양제 에 나아가서 끓어앉아 나아가서 선다 판통례가 , 다만 술 서문 궤 앞으로 나아가  
부복하고 끓어앉아 부복

533/533 - 250s - loss: 0.4581 - accuracy: 0.9256

---

먼저 모델을 학습시키면서 모델의 생성 결과물을 확인하기 위해 testmodel이라는 이름으로 콜백 함수를 정의했습니다.

---

```
test_text_X = test_sentence.split(' ')[-seg_length:]
```

---

먼저 임의의 문장을 입력한 다음, 뒤에서부터 seg\_length만큼의 단어(25개)를 선택합니다.

---

```
test_text_X = np.array([word2idx[c] if c in word2idx else word2idx['UNK'] for c in test_text_X])
```

---

그다음에는 문장의 단어를 인덱스 토큰으로 바꿉니다. 이때 사전에 등록돼 있지 않은 단어의 경우에는 'UNK' 토큰 값으로 바꿉니다.

---

```
test_text_X*= pad_sequences([test_text_X], maxlen=seq_length, padding='pre',  
value=word2idx['UNK'])
```

---

그다음에 7.3절에서 사용했던 pad\_sequences()로 문장의 앞쪽의 빈 자리가 있을 경우 25단어가 채워지도록 패딩을 넣습니다. 이때 패딩 값인 value 인수에는 앞에서 지정했던 'UNK' 토큰의 값인 word2idx['UNK']를 사용합니다.

---

```
output_idx = model.predict_classes(test_text_X)
```

---

`model.predict()`는 Dense 레이어의 출력값인 332,640개의 값을 반환하기 때문에 출력을 간결하게 하기 위해 `model.predict_classes()` 함수를 사용합니다. 이 함수는 출력 중에서 가장 값이 큰 인덱스를 반환합니다.

```
test_sentence += ' ' + idx2word[output_idx[0]]
```

출력 단어는 `test_sentence`의 끝부분에 저장되어 다음 스텝의 입력에 활용됩니다.

이렇게 정의된 콜백 함수는 `testmodelcb`라는 이름으로 저장되어 `tf.keras`의 `model.fit()`의 `callbacks` 인수에 포함됩니다.

```
history = model.fit(train_dataset.repeat(), epochs=50, steps_per_epoch=steps_per_epoch,
 callbacks=[testmodelcb], verbose=2)
```

학습을 시킬 때도 위에서 정의한 `Dataset`를 활용합니다. 그런데 주목해야 할 점은 `Dataset`에서 데이터를 끊임없이 반환하도록 `repeat()` 함수를 사용해야 한다는 점입니다. 그리고 넘파이 array 형태의 데이터를 사용할 때는 에포크마다 데이터를 한번씩 순회시켰지만, `Dataset`를 사용하면 데이터의 시작과 끝을 알 수 없기 때문에 에포크에 데이터를 얼마나 학습시킬지를 `steps_per_epoch` 인수로 지정해야 합니다.

학습 결과를 살펴보면 처음에는 의미없는 단어가 반복되지만 학습을 진행하면 점점 부분적으로 문맥이 연결되는 단어들이 배열되는 것을 알 수 있습니다. 더 많이 학습시킬수록 더 자연스러운 결과를 얻게 됩니다.

이제 임의의 문장을 넣어서 학습이 잘 됐는지 알아보겠습니다. 난중일기의 한 구절을 입력해 보겠습니다.

예제 7.42

임의의 문장을 사용한 생성 결과 확인

[IN]

```
from tensorflow.keras.preprocessing.sequence import pad_sequences
test_sentence = '동헌에 나가 공무를 본 후 활 십오 순을 쏘았다'

next_words = 100
for _ in range(next_words):
```

```

test_text_X = test_sentence.split(' ')[:-seq_length:]
test_text_X = np.array([word2idx[c] if c in word2idx else total_words+1 for c in
test_text_X])
test_text_X = pad_sequences([test_text_X], maxlen=seq_length, padding='pre' ,
value=word2idx['UNK'])

output_idx = model.predict_classes(test_text_X)
test_sentence += ' ' + idx2word[output_idx[0]]

print(test_sentence)

```

#### [OUT]

동현에 나가 공무를 본 후 활 십오 순을 쏘았다 그곳의 사로잡힌 대리하는 호군직 2개는 놓기도 토의  
으로써 차임 되고 , 저들은 일시에 굳게 것으로서 주고 , 더욱 높은 바가 많아서 정위 라 같다 하는데  
하였습니다 빙자 는 부의 에 도와서 알아야 하여 주소서

하니 , 형조 에서 교지 를 하사하고 충청도 도절제사 에서 다시 와서 와서 내려 쓸 하고 , 만일 잘 가는  
것이 없었다 상수의 대한 한 한 뒤에 모두 능히 서로 알지 것은 감히 모두 모두 모두 사람을 죄를 죄를  
사람을 죄를 온 한 사람을 삼가서 어질고 그믐날에 한다는 것이니 , 청컨대 이러한 정성을 알아서 난 를  
정하여 급히 원하는 곳에 두고 특별히 징계하여 인도해 특별히

전체적인 문장의 의미가 잘 통하지는 않지만 부분 부분에서는 자연스럽게 연결되는 단어들이 보이는 것  
을 확인할 수 있습니다.

## 7.4.2 자소 단위 생성

자소 단위 생성을 하기 위해서는 한글을 자소 단위로 분리하고 다시 합칠 수 있는 라이브러리가 필요합  
니다. 이러한 작업을 할 수 있는 라이브러리로 신해빈 님이 만든 jamotools<sup>26</sup>가 있습니다. 배시 셀 명령어  
를 이용해 라이브러리를 설치해보겠습니다.

**예제 7.43** jamotools 설치

#### [IN]

```
!pip install jamotools
```

<sup>26</sup> <https://github.com/HaebinShin/jamotools>

[OUT]

```
Collecting jamotools
  Downloading https://files.pythonhosted.org/packages/3d/d6/ec13c68f7ea6a8085966390d256d183bf
8488f8b9770028359acb86df643/jamotools-0.1.10-py2.py3-none-any.whl
Requirement already satisfied: future in /usr/local/lib/python3.6/dist-packages (from
jamotools) (0.16.0)
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages (from jamotools)
(1.12.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages (from
jamotools) (1.16.4)
Installing collected packages: jamotools
Successfully installed jamotools-0.1.10
```

jamotools의 기능을 테스트해보기 위해 조선왕조실록 텍스트를 다시 사용하겠습니다.

### 예제 7.44 자모 분리 테스트

[IN]

```
import jamotools

train_text = open(path_to_file, 'rb').read().decode(encoding='utf-8')
s = train_text[:100]
print(s)
```

# 한글 텍스트를 자모 단위로 분리합니다. 한자 등에는 영향이 없습니다.

```
s_split = jamotools.split_syllables(s)  
print(s_split)
```

[OUT]

태조 이성계 선대의 가계. 목조 이안사가 전주에서 삼척·의주를 거쳐 알동에 정착하다  
태조 강현 지인 계운 성문 신무 대왕(太祖康獻至仁啓運聖文神武大王)의 성은 이씨(李氏)요, 휘  
이우(惟)로, 본관은 경북(慶北), 자는 우경(禹卿), 호는 청당(淸堂)이다.  
태조는 그의 아버지 이언(李彦)과 어머니 이씨(李氏)의 이름을 따서 태어난 것이다.  
태조는 그의 아버지 이언(李彦)과 어머니 이씨(李氏)의 이름을 따서 태어난 것이다.

`split_syllables()` 함수를 이용해 100글자의 한글이 자모 단위로 정상적으로 분리되는 것을 확인할 수 있습니다. `jamotools`는 영문이나 한자 등에는 영향을 주지 않습니다. 분리한 자모를 다시 합칠 수도 있습니다.

## [IN]

```
s2 = jamotools.join_jamos(s_split)
print(s2)
print(s == s2)
```

## [OUT]

```
태조 이성계 선대의 가계. 목조 이안사가 전주에서 삼척·의주를 거쳐 알동에 정착하다
태조 강현 지인 계운 성문 신무 대왕(太祖康獻至仁啓運聖文神武大王)의 성은 이씨(李氏)요, 휘
True
```

자모를 분리했다가 다시 합친 s2는 원래 텍스트인 s와 같다는 것을 두 번째 출력이 True인 것에서 확인할 수 있습니다.

그럼 이제 자모를 토큰화해보겠습니다. 여기서는 따로 텍스트 전처리를 하지 않을 예정이기 때문에 괄호, 한자 등이 토큰에 모두 포함될 것입니다.

## [IN]

```
# 텍스트를 자모 단위로 나눕니다. 데이터가 크기 때문에 약간 시간이 걸립니다.
train_text_X = jamotools.split_syllables(train_text)
vocab = sorted(set(train_text_X))
vocab.append('UNK')
print ('{} unique characters'.format(len(vocab)))

# vocab list를 숫자로 매핑하고, 반대도 실행합니다.
char2idx = {u:i for i, u in enumerate(vocab)}
idx2char = np.array(vocab)

text_as_int = np.array([char2idx[c] for c in train_text_X])

# word2idx의 일부를 알아보기 쉽게 출력해 봅니다.
print('{}')
for char,_ in zip(char2idx, range(10)):
    print(' {:4s}: {:3d}'.format(repr(char), char2idx[char]))
print(' ...\\n')

print('index of UNK: {}'.format(char2idx['UNK']))
```

### [OUT]

```
6198 unique characters
```

```
{  
    '\n': 0,  
    ' ': 1,  
    '!': 2,  
    '\"': 3,  
    '\"': 4,  
    '(': 5,  
    ')': 6,  
    '+': 7,  
    ',': 8,  
    '-': 9,  
    ...  
}
```

```
index of UNK: 6197
```

자모를 토큰화하고 혹시 사전에 정의되지 않은 기호를 만날 수도 있으므로 'UNK'도 사전에 넣습니다. 중복되지 않는 자모는 총 6,198개가 나옵니다. 단어보다는 훨씬 적은 양이고, 한자 등을 뺐다면 더 줄어들었을 것입니다만 여기서는 조선왕조실록 텍스트의 형태를 비슷하게 따라 하는 결과물을 만드는 것이 목적이기 때문에 한자 등을 빼지 않았습니다.

토큰 데이터를 출력해보겠습니다.

### 예제 7.47 토큰 데이터 확인

### [IN]

```
print(train_text_X[:20])  
print(text_as_int[:20])
```

### [OUT]

```
ㅌ ㅎ ㅈ ㅗ ㅇ | ㅅ ㅏ ㅇ ㄱ ㅋ ㅌ ㅅ ㅓ ㄴ ㄷ ㅐ ㅎ ㅇ  
[6158 83 87 79 94 1 78 106 76 90 78 56 93 1  
76 90 59 62 87 78]
```

'ㅌ'은 6158, 'ㅓ'는 83 등으로 토큰화된 것을 확인할 수 있습니다.

그다음으로는 단어 단위 생성에서 했던 것처럼 학습 데이터세트를 생성해야 합니다. 이 부분의 코드는 큰 변경사항이 없습니다.

### 예제 7.48 학습 데이터세트 생성

[IN]

```
seq_length = 80
examples_per_epoch = len(text_as_int) // seq_length
char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)

char_dataset = char_dataset.batch(seq_length+1, drop_remainder=True)
for item in char_dataset.take(1):
    print(idx2char[item.numpy()])
    print(item.numpy())

def split_input_target(chunk):
    return [chunk[:-1], chunk[-1]]

train_dataset = char_dataset.map(split_input_target)
for x,y in train_dataset.take(1):
    print(idx2char[x.numpy()])
    print(x.numpy())
    print(idx2char[y.numpy()])
    print(y.numpy())

BATCH_SIZE = 256
steps_per_epoch = examples_per_epoch // BATCH_SIZE
BUFFER_SIZE = 10000

train_dataset = train_dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)
```

[OUT]

단어 단위에서 25개의 단어를 입력받았을 때 1개의 단어를 출력했던 것에 비해 자소 단위에서는 80개의 자소를 입력받았을 때 1개의 자소를 출력하도록 seq\_length를 80으로 잡았습니다.

이제 자소 단위의 생성 모델을 정의해 보겠습니다.

#### 예제 7.49 자소 단위 생성 모델 정의

[IN]

```
total_chars = len(vocab)
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(total_chars, 100, input_length=seq_length),
    tf.keras.layers.LSTM(units=400),
    tf.keras.layers.Dense(total_chars, activation='softmax')
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

[OUT]

Model: "sequential"

Layer (type)	Output Shape	Param #
--------------	--------------	---------

embedding (Embedding)	(None, 80, 100)	619800
unified_lstm (UnifiedLSTM)	(None, 400)	801600
dense (Dense)	(None, 6198)	2485398

Total params: 3,906,798  
 Trainable params: 3,906,798  
 Non-trainable params: 0

자소 단위 생성 모델에서는 겹쳐진 순환 신경망을 사용하지 않고 LSTM 레이어를 하나만 사용했습니다. 대신 하나의 LSTM 레이어에서 사용하는 뉴런의 수를 4배로 늘렸습니다. 또 단어의 수보다 자소의 수가 훨씬 적기 때문에 마지막 Dense 레이어의 뉴런 수가 적어졌습니다.

다른 부분은 단어 단위 생성 모델과 같습니다. 그럼 이제 실제로 모델을 학습시켜보겠습니다.

#### 예제 7.50 자소 단위 생성 모델 학습

[IN]

```
from tensorflow.keras.preprocessing.sequence import pad_sequences

def testmodel(epoch, logs):
    if epoch % 5 != 0 and epoch != 99:
        return

    test_sentence = train_text[:48]
    test_sentence = jamotools.split_syllables(test_sentence)

    next_chars = 300
    for _ in range(next_chars):
        test_text_X = test_sentence[-seq_length:]
        test_text_X = np.array([char2idx[c] if c in char2idx else char2idx['UNK'] for c in
test_text_X])
        test_text_X = pad_sequences([test_text_X], maxlen=seq_length, padding='pre',
value=char2idx['UNK'])

    output_idx = model.predict_classes(test_text_X)
```

```

    test_sentence += idx2char[output_idx[0]]

    print()
    print(jamotools.join_jamos(test_sentence))
    print()

testmodelcb = tf.keras.callbacks.LambdaCallback(on_epoch_end=testmodel)

history = model.fit(train_dataset.repeat(), epochs=100, steps_per_epoch=steps_per_epoch,
callbacks=[testmodelcb], verbose=2)

```

### [OUT]

Epoch 1/100

태조 이성계 선대의 가계. 목조 이안사가 전주에서 삼척 · 의주를 거쳐 알동에 정착하다 이를 것을  
 것을 것을 것을 것을 것을 것을 것을 것을 것을 것을 것을 것을 것을 것을 것을 것을 것을  
 것을 것을 것을 것을 것을 것을 것을 것을 것을 것을 것을 것을 것을 것을 것을 것을  
 것을 것을 것을 것을 것을 것을

2364/2364 - 270s - loss: 2.5904 - accuracy: 0.3065

(생략)

Epoch 21/100

태조 이성계 선대의 가계. 목조 이안사가 전주에서 삼척 · 의주를 거쳐 알동에 정착하다 하는 것은  
 그 고을에 있으면 그 사람을 감하게 하였다. 임금이 말하기를,  
 "각각 15석을 가지고 감사(監司)에서 아뢰기를,  
 "각각 15석을 가지

2364/2364 - 275s - loss: 1.2607 - accuracy: 0.5970

(생략)

Epoch 41/100

태조 이성계 선대의 가계. 목조 이안사가 전주에서 삼척 · 의주를 거쳐 알동에 정착하다 한다.  
 전하께서는 알지 못한 자가 있으면 나와서 아뢰다  
 예조에서 아뢰기를,  
 "이 앞서 있는 것이 어떻게 아뢰다  
 예조에서 아뢰기를,  
 "이 앞서 정지는 알지 못한 자가 있으면 날이 없으니, 이제 있는 것이 어떻겠습니까. 이제

상소하기를,  
"이 앞서 중에 있는 것은 알지 못한 작

2364/2364 - 268s - loss: 1.0895 - accuracy: 0.6494  
(생략)  
Epoch 61/100

태조 이성계 선대의 가계. 목조 이안사가 전주에서 삼척·의주를 거쳐 알동에 정착하다 하여  
바라옵건대, 상소(上疏)하여 상고하여 아뢰기를,  
"이 사람은 반드시 사이에 있었으니, 이것을 아뢰다  
의정부에서 아뢰기를,  
"이 사람은 임금이 말하기를, '고을의 각 고을의 성을 일으키고 돌아왔으나, 이는 이름을 인도하여  
아뢰기를,  
"성상의 의논을 인도하여 아뢰기를,  
"이 ○

2364/2364 - 272s - loss: 0.9945 - accuracy: 0.6779  
(생략)  
Epoch 100/100

태조 이성계 선대의 가계. 목조 이안사가 전주에서 삼척·의주를 거쳐 알동에 정착하다 하여, 일이  
있으면, 중국의 음사운의 청한 것은 아뢰기를,  
"각도의 아들을 보내어 여러 신하들이 이르기를, '군사의 임금이 말하기를,  
"이박한 그렇게 한 것은 아뢰기를,  
"각도에 이르렀던 것이 없습니다. 그러나 이에 가서 수령을 감동하다  
사현부에서 계하기를,  
"원산군(楊根

2364/2364 - 274s - loss: 0.9206 - accuracy: 0.6999

단어 생성 모델과 같은 문장을 자소 단위로 넣어서 에포크가 끝날 때마다 생성 결과를 확인하게 했습니다. 자소 단위 생성 모델의 accuracy는 단어 단위 생성 모델과 마찬가지로 꾸준히 올라갑니다. 단어 생성 모델과 마찬가지로 처음에는 반복되는 패턴이 자주 나타나지만 점점 그럴듯한 결과를 만들어내기 시작합니다. 특히 한자와 팔호를 그대로 학습에 사용하고 있기 때문에 "감사(監司)", "상소(上疏)" 같은 한글과 한자의 병기도 잘 해내는 것을 볼 수 있습니다.

이제 임의의 문장을 넣어서 학습이 잘 됐는지 확인해 보겠습니다.

**[IN]**

```

from tensorflow.keras.preprocessing.sequence import pad_sequences
test_sentence = '동현에 나가 공무를 본 후 활 십오 순을 쏘았다'
test_sentence = jamotools.split_syllables(test_sentence)

next_chars = 300
for _ in range(next_chars):
    test_text_X = test_sentence[-seq_length:]
    test_text_X = np.array([char2idx[c] if c in char2idx else char2idx['UNK'] for c in test_text_X])
    test_text_X = pad_sequences([test_text_X], maxlen=seq_length, padding='pre',
                                value=char2idx['UNK'])

    output_idx = model.predict_classes(test_text_X)
    test_sentence += idx2char[output_idx[0]]


print(jamotools.join_jamos(test_sentence))

```

**[OUT]**

동현에 나가 공무를 본 후 활 십오 순을 쏘았다. 임금이 말하기를,  
 "이보다 큰 공상은 그 집에 돌아오다  
 정사를 보았다. 임금이 말하기를,  
 "이방성을 아뢰다  
 함길도 감사가 이미 나라를 행하였다. 상왕이 그 사람을 금하다  
 임금이 말하기를,  
 "이보다 큰 공상은 그 집에 돌아온다. 【모든 것을 보내어 여러 관원은 농산】

여기서도 뒤로 가면 비슷한 문장이 다시 나오고 있지만 주목해야 할 점은 자소 단위로 학습을 시켰음에도 불구하고 한글을 정확하게 조합할 수 있도록 네트워크가 자소를 생성하고 있다는 점입니다.

## 7.5 정리

이번 장에서는 변화하는 입력에 대해 같은 레이어를 사용해 결과를 얻어내는 순환 신경망에 대해서 알아봤습니다. 순환 신경망의 주요 레이어로는 SimpleRNN, LSTM, GRU 레이어가 있었고 순환 신경망의 대표적인 용도 중 하나인 자연어 처리에 쓰이는 임베딩 레이어도 살펴봤습니다.

주요 예제로는 LSTM 논문에서 사용됐던 곱셈 문제와 긍정/부정 감성 분석, 자소 단위와 단어 단위 자연어 생성을 코드와 함께 확인해봤습니다.

## 사전 훈련된 모델 다루기

딥러닝이 발전함에 따라 다양한 네트워크가 개발됐습니다. 좋은 성능을 보이는 네트워크는 수십, 수백 개의 레이어를 쌓은 경우가 대부분이고, 레이어가 늘어남에 따라 네트워크를 훈련시키는 데 걸리는 시간도 증가합니다. 유명한 CNN 중 하나인 ResNet-50은 8장의 P100 GPU를 사용해 ImageNet의 사진을 잘 분류하도록 학습시키는 데 29시간이 걸립니다. 페이스북 연구팀은 이 시간을 1시간으로 줄였지만 그 대신 훨씬 많은 256장의 GPU를 사용했습니다.<sup>1</sup> 최근 이미지 분류 문제에서 더 좋은 결과를 내고 있는 NASNet은 AutoML 기법<sup>2</sup>을 사용해 최적의 네트워크 구조를 스스로 학습하기 때문에 훨씬 더 많은 훈련 시간이 필요합니다.<sup>3</sup>

다행히 딥러닝 기술은 개방적인 환경에서 빠르게 발전하고 있습니다. 연구자들은 자신이 만든 사전 훈련된 모델(pre-trained model)을 인터넷에 올려놓아 다른 사람들이 쉽게 내려받을 수 있게 합니다. 이렇게 얻은 모델을 그대로 사용할 수도 있고, 전이 학습(Transfer Learning)이나 신경 스타일 전이(Neural Style Transfer)처럼 다른 과제를 위해 재가공해서 사용할 수도 있습니다.

### 8.1 텐서플로 허브

텐서플로에서 제공하는 텐서플로 허브(TensorFlow Hub)는 재사용 가능한 모델을 쉽게 이용할 수 있는 라이브러리입니다. 텐서플로 허브 홈페이지에서는 이미지, 텍스트, 비디오 등의 분야에서 사전 훈련된 모델들을 검색해볼 수 있습니다.

<sup>1</sup> Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He, Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, <https://arxiv.org/abs/1706.02677>

<sup>2</sup> Automated Machine Learning의 약자로 알고리즘 선택, 모델의 하이퍼 파라미터 튜닝, 반복 모델링, 모델 평가를 자동으로(마신러닝으로) 수행하는 방법론입니다.

<sup>3</sup> Zoph, B., Vasudevan, V., Shlens, J., Le, Q.V.: Learning transferable architectures for scalable image recognition, CVPR 2018, <https://arxiv.org/abs/1707.07012>. NASNet 논문에서는 450장의 GPU를 사용해 20,000개의 모델을 만들어낼 때까지 탐색을 수행합니다.

The screenshot shows the TensorFlow Hub homepage. On the left, there's a sidebar with 'Quick links' to Home, All collections, All models, and All publishers. Below that are sections for 'Problem domains' (Image, Text, Video) and 'Support' (Intro to TF Hub, Intro to ML). The main content area has a search bar at the top. A large orange header says 'Hello. Welcome to TensorFlow Hub.' Below it, a message reads: 'Whether you're publishing or browsing, this repository is where hundreds of machine learning models come together in one place. Thanks for playing a part in our community.' There are two main calls-to-action: 'Discover our hub' (with a 'Get started' button) and 'Meet our community' (with a 'Let's go' button).

그림 8.1 텐서플로 허브 홈페이지<sup>4</sup>

텐서플로 2.0을 사용하면 텐서플로 허브는 따로 설치할 필요가 없고 라이브러리를 바로 불러올 수 있습니다.<sup>5</sup>

예제 8.1 텐서플로 허브에서 사전 훈련된 MobileNet 모델 불러오기

[IN]

```
import tensorflow_hub as hub

mobile_net_url = "https://tfhub.dev/google/tf2-preview/mobilenet_v2/classification/2"
model = tf.keras.Sequential([
    hub.KerasLayer(handle=mobile_net_url, input_shape=(224, 224, 3), trainable=False)
])
model.summary()
```

4 <https://tfhub.dev/>

5 텐서플로 1.x 버전을 사용한다면 !pip install tensorflow-hub 명령으로 따로 설치해야 합니다.

## [OUT]

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
keras_layer_1 (KerasLayer)	None	3540265
Total params: 3,540,265		
Trainable params: 0		
Non-trainable params: 3,540,265		

먼저 tensorflow\_hub 라이브러리를 hub라는 약어로 임포트합니다. 그다음 컨볼루션 신경망의 하나인 MobileNet 버전 2<sup>6</sup>를 불러옵니다. MobileNet은 계산 부담이 큰 컨볼루션 신경망을 연산 성능이 제한된 모바일 환경에서도 작동 가능하도록 네트워크 구조를 경량화한 것입니다. MobileNet 버전 2는 버전 1을 개선했고 파라미터 수도 더 줄어들었습니다.

MobileNet 버전 2는 그림 8.2에서 볼 수 있듯이 복잡한 네트워크 구조를 가지고 있습니다.

<sup>6</sup> 『MobileNetV2: The Next Generation of On-Device Computer Vision Networks』, 구글 AI 블로그, <http://bit.ly/2IVwU9A>

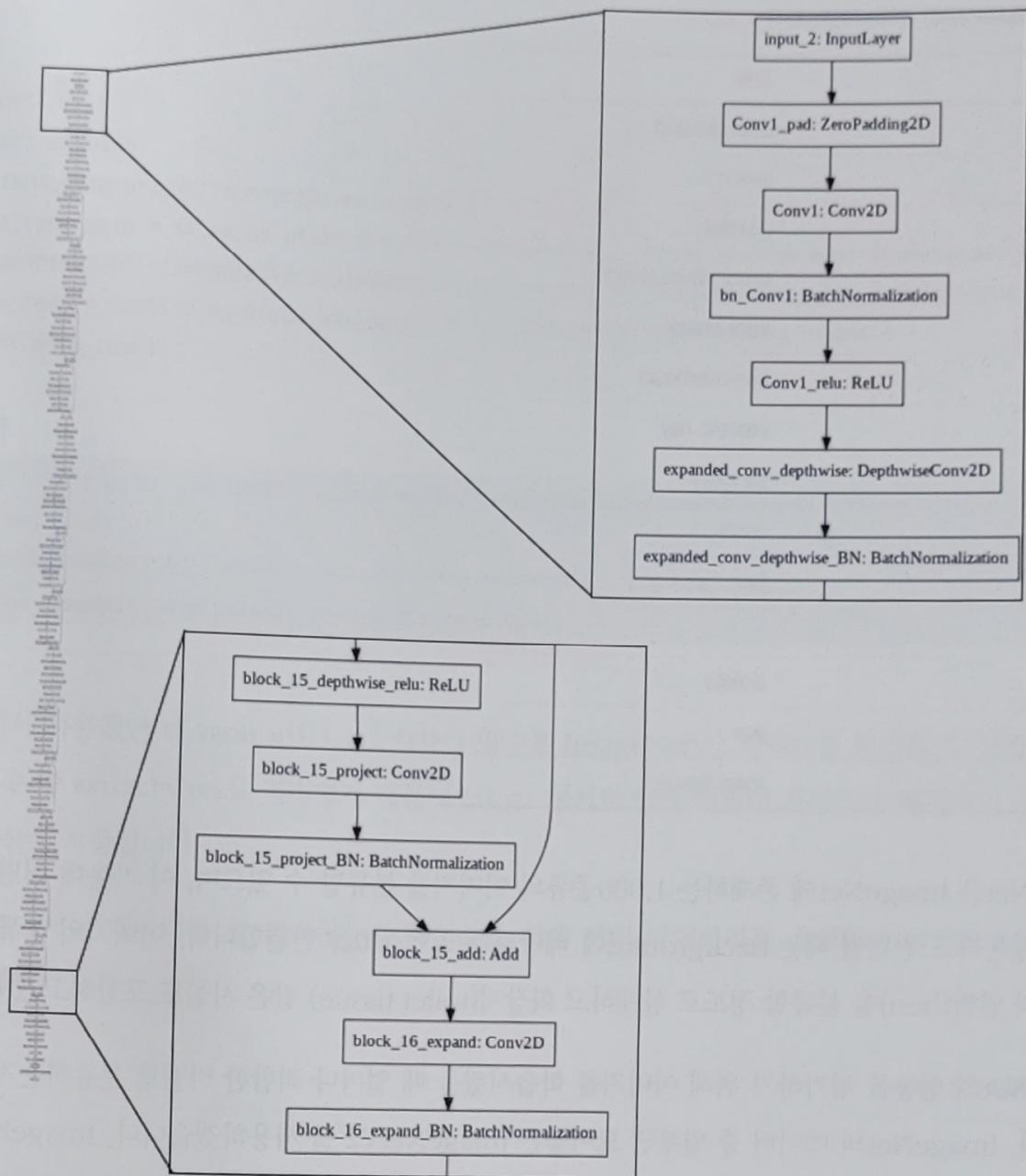


그림 8.2 MobileNet v2의 구조. 레이어 수는 총 157개입니다.

텐서플로 허브에 올라와 있는 모델은 `hub.KerasLayer()` 명령으로 `tf.keras`에서 사용 가능한 레이어로 변환할 수 있습니다. `model.summary()`로 파라미터 개수를 확인해보면 354만 개 정도입니다. ResNet-50에는 2,560만 개, ResNet-152에는 6천만 개의 파라미터가 존재하는 것에 비교하면 상대적으로 파라미터 수가 적은 편입니다.

예제 8.1에서 불러온 MobileNet은 ImageNet 데이터로 학습시켰습니다. 학습된 이미지의 분류 중 일부는 표 8.1에 나와 있습니다.