06

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 또다른 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.

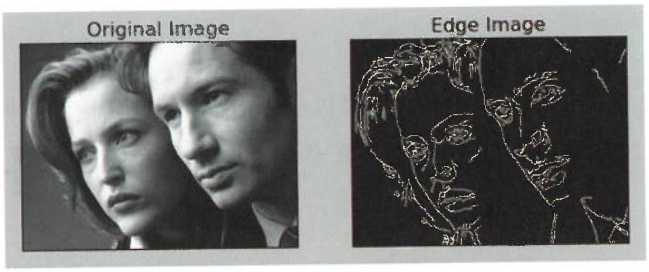


그림 6.1 외곽선 검출 알고리즘 중 하나인 Canny Edge Detection을 이용한 결과 이미지[1]

이런 특징 추출(Feature Extraction) 기법 중 하나인 컨볼루션 연산은 각 픽셀을 본래 픽셀과 그 주변 픽셀의 조합으로 대체하는 동작[2]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.

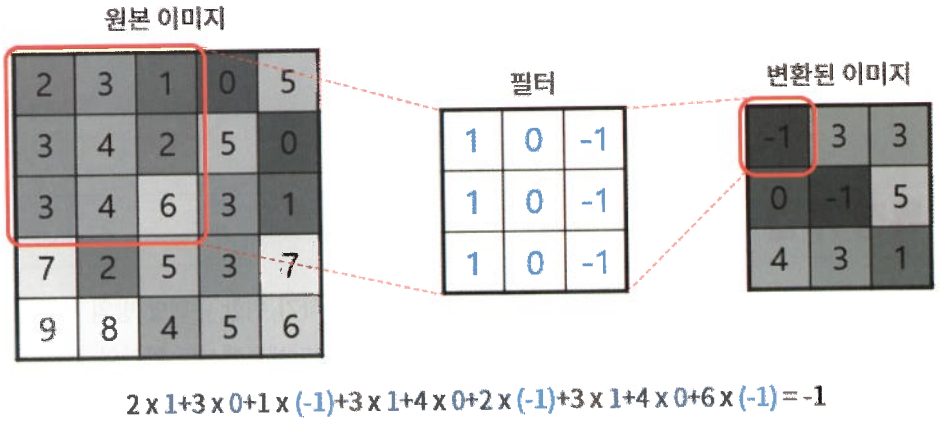


그림 6.2 컨볼루션 연산은 원본 이미지와 필터 행렬의 합성곱입니다.

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

2 네이버 IT 용어사전: http://bit.ly/2XlMBTG

간단한 컨볼루션 필터 몇 가지를 소개해 보겠습니다. 그림 6.3에서 쓰인 3×3 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[3]

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.[4] 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 문제점들을 모두 해결했습니다. 컨볼루션 신경망은 특징을 검출하는 필터를 수작업으로 설계하는 것이 아니라 **네트워크가 특징을 추출하는 필터를 자동으로 생성**합니다. 학습을 계속하면 네트워크를 구성하는 각 뉴런들은 입력한 데이터에 대해 특정 패턴을 잘 추출할 수 있도록 적응하게 됩니다.

3 더 다양한 컨볼루션 필터는 다음 링크에서 찾아볼 수 있습니다. https://en.wikipedia.org/wiki/Kernel_(image_processing)

4 Kai Yu & Andrew Ng, 《Feature Learning for Image Classification》, ECCV2010, http://bit.ly/2Lfzxyx

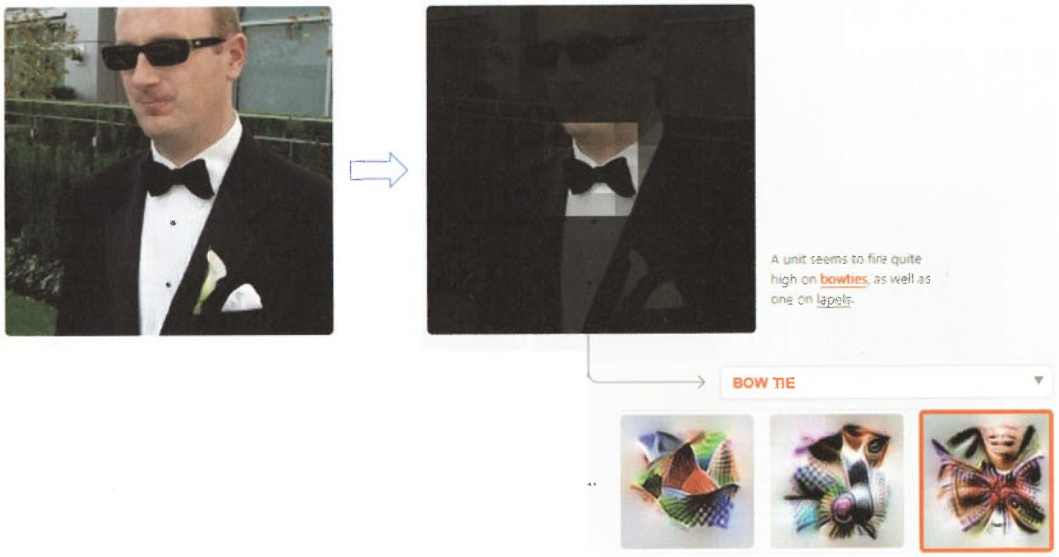


그림 6.4 나비 넥타이에 반응하는 뉴런(우측 하단)은 이미지의 나비 넥타이를 정확히 찾습니다.[5]

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.

5 https://distill.pub/2018/building-blocks/

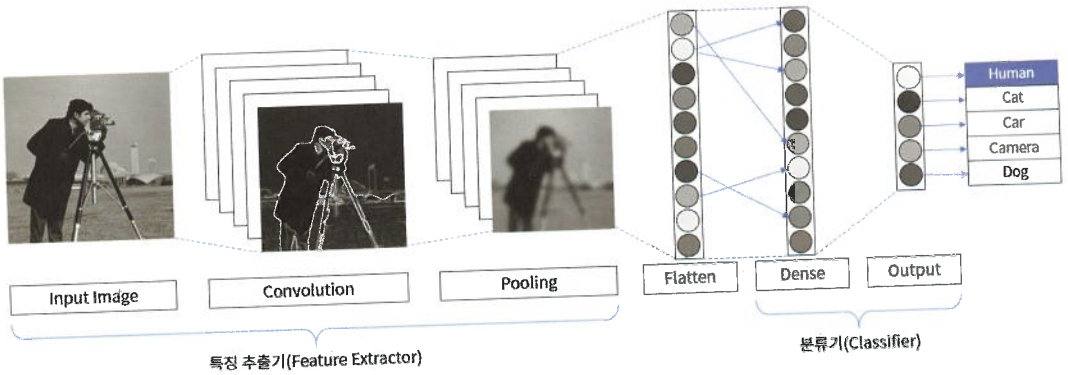


그림 6.5 이미지 분류에 사용되는 컨볼루션 신경망의 구조[6]

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

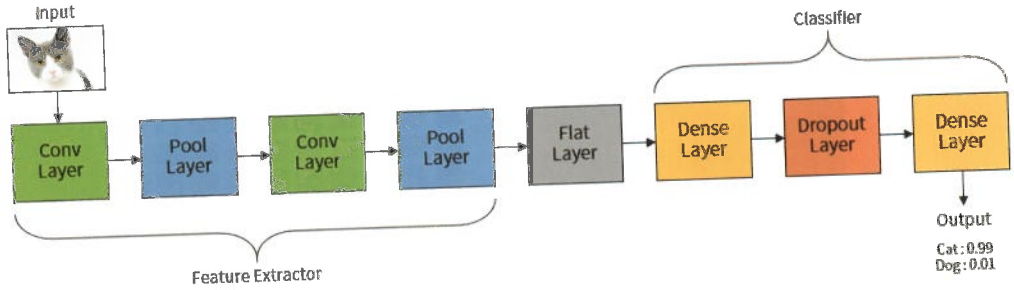


그림 6.6 이미지 분류에 사용되는 컨볼루션 신경망의 구조

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

6 출처: Convolutional Neural Networks for Image Classification – General Architecture of a Convolutional Neural Network, completegate. 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.

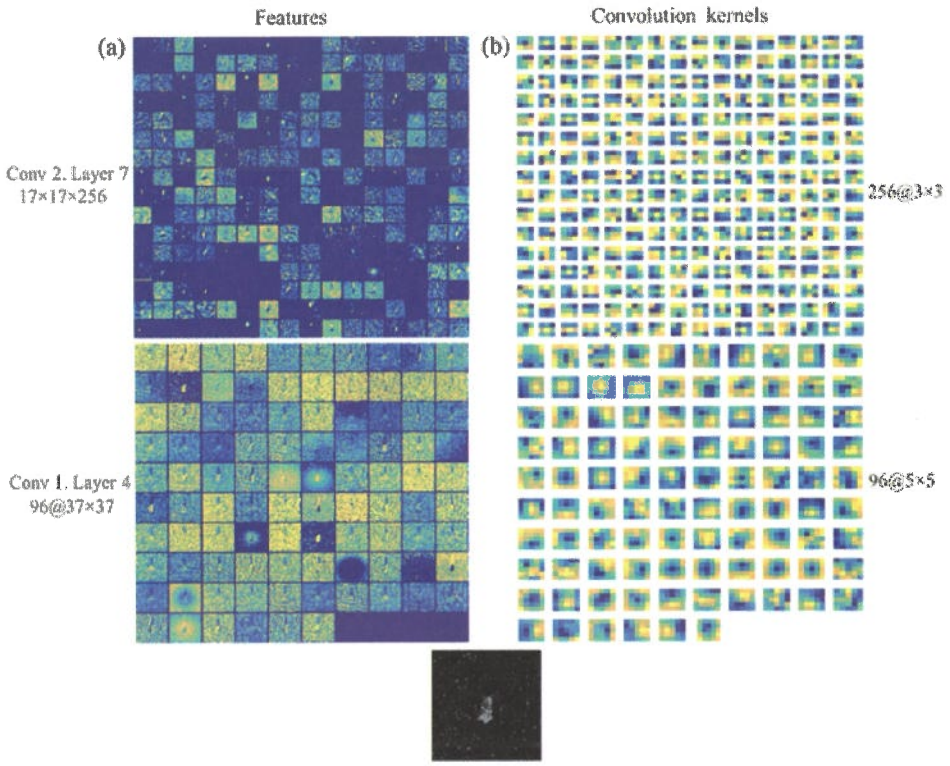


그림 6.7 오른쪽은 필터, 왼쪽은 필터로 추출한 중간 계산 이미지(특징)입니다.[7]

7 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.

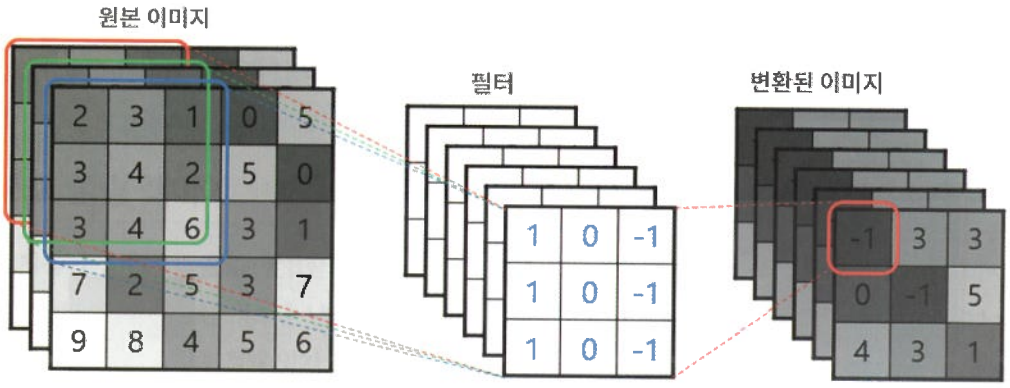


그림 6.9 RGB 채널을 가진 이미지에 컨볼루션 연산을 적용한 결과

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, filters의 네 가지입니다.

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

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

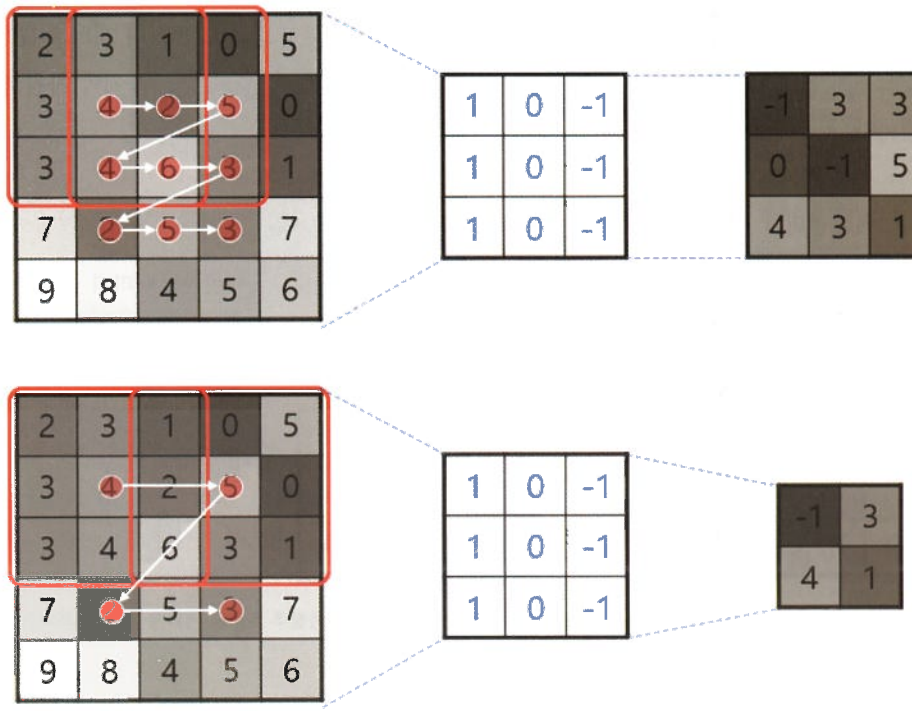


그림 6.10 strides=1일 때와 strides=2일 때의 결과 이미지 비교. 붉은색 원은 필터의 중심

padding은 컨볼루션 연산 전에 입력 이미지 주변에 빈 값을 넣을지 지정하는 옵션으로서 'valid'와 'same'이라는 2가지 옵션 중 하나를 사용합니다. 'valid'는 우리가 봐온 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.

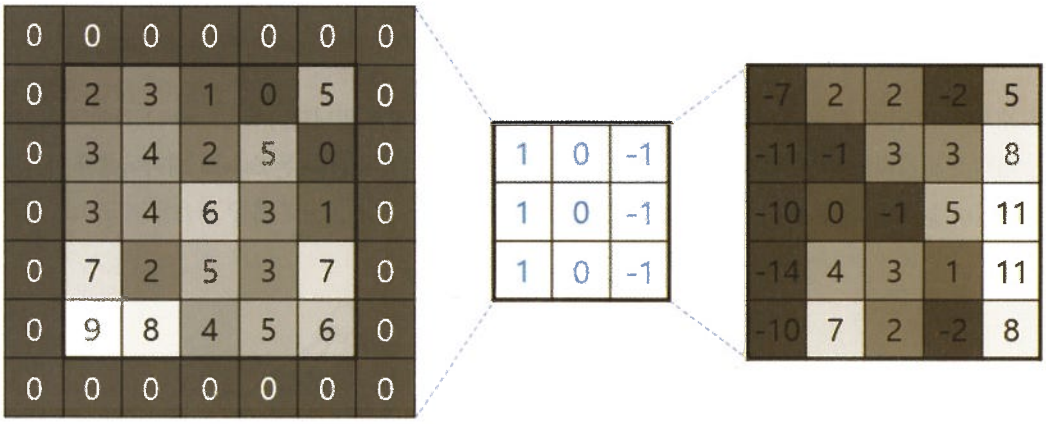


그림 6.11 padding='same'일 때 입력 이미지와 출력 이미지의 크기가 같아집니다.

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 → 128 → 256 → 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.

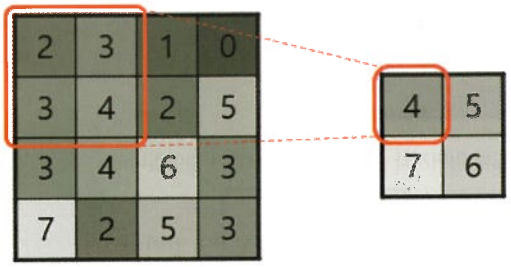


그림 6.12 pool_size=(2,2), strides=(2,2)일 때의 MaxPool2D 레이어 적용 결과

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.[8] 레딧(reddit)의 머신러닝 커뮤니티에서

8 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[9]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.*[10]

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

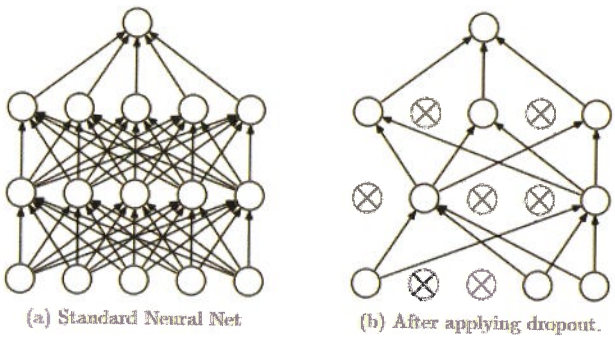


그림 6.13 드롭아웃 레이어의 동작 구조[11]

드롭아웃 레이어는 앞에서 인용한 것처럼 학습 과정에서 무작위로 뉴런의 부분집합을 제거하는 것입니다. 네트워크가 학습할 때 같은 레이어에 있는 뉴런들은 결괏값에 의해 서로 같은 영향을 받습니다. 따라서 각 뉴런의 계산 결과를 모두 더해서 나오는 결괏값은 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.

9 http://bit.ly/2XExLO1
10 http://bit.ly/2LWaIXM
11 Srivastava, Nitish, et al. Dropout: a simple way to prevent neural networks from overfitting. JMLR 2014

예제 6.2.3 드롭아웃 레이어 생성 코드

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

주요 인수는 rate로, 제외할 뉴런의 비율을 나타냅니다. 간단한 레이어이지만 AlexNet, VGG, GoogLeNet, DenseNet 등 거의 모든 주요 컨볼루션 신경망에 사용됐습니다. 드롭아웃 레이어도 가중치가 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.

예제 6.4 Fashion MNIST 데이터세트 불러오기 및 정규화

```
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. 60000×28×28과 60000×28×28×1이 같은 값인 것처럼 말입니다.

데이터를 확인하기 위해 matplotlib.pyplot을 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 취해 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

라벨	범주
0	티셔츠/상의
1	바지
2	스웨터
3	드레스
4	코트
5	샌들
6	셔츠
7	운동화
8	가방
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 발하는데, 이 책에는 그 정도의 큰 예제는 다루지 않으므로 여기서는 GPU를 사용하겠습니다. GPU로 설정한 뒤에는 우측 하단의 '저장' 버튼을 눌러서 해당 설정을 적용합니다.

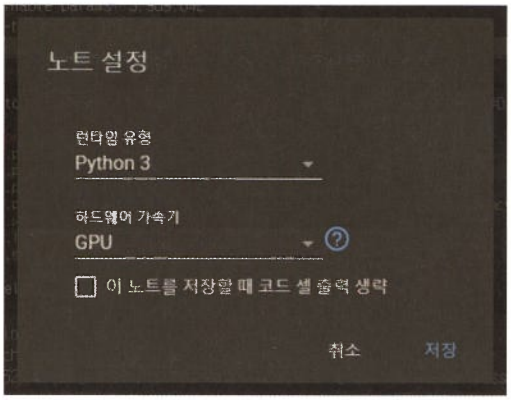


그림 6.14 구글 코랩에서 코드를 실행 전에 하드웨어 가속기를 GPU로 설정합니다.

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 |                    0 |
| N/A   29C    P8    27W / 149W |      0MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

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

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.[12]

```
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]

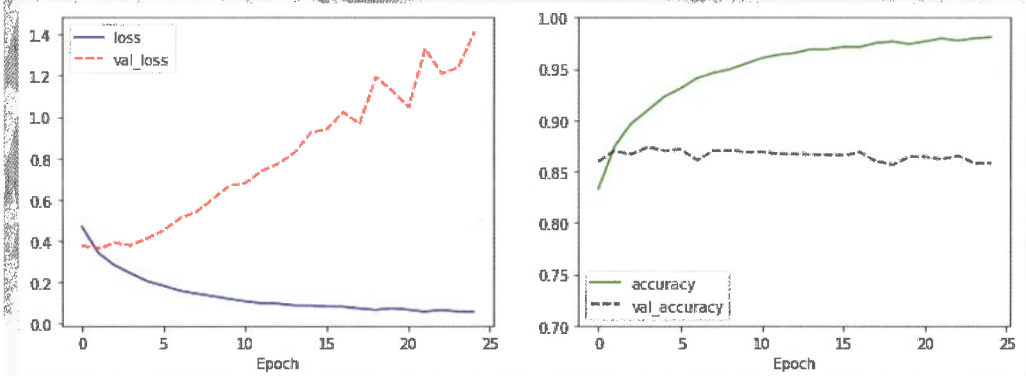
```
history = model.fit(train_X, train_Y, epochs=25, validation_split=0.25)

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()
```

12 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)
```




```
[1.4777660627841949, 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 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (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.

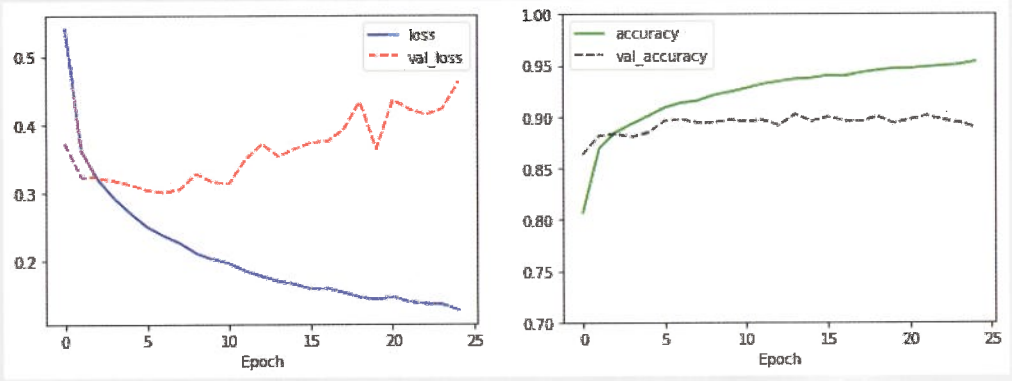
[IN]

```
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. 풀링 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.[13] 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.

13 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.

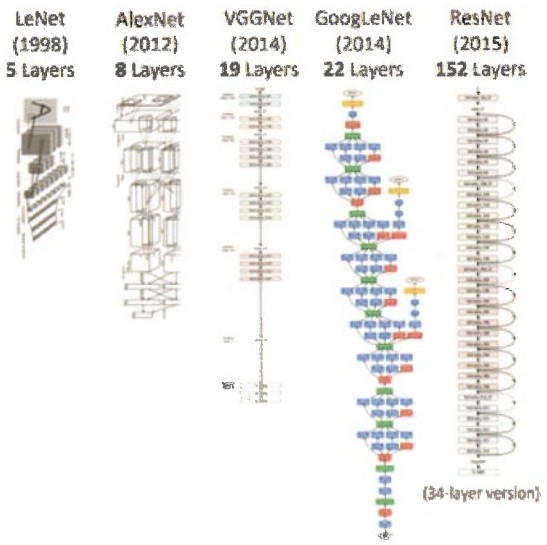


Figure 1. Progression towards deeper neural network structures in recent years (see, e.g., [6], [7], [8], [9], [10]).

그림 6.15 점점 깊어진 컨볼루션 신경망[14]

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[15]에서도 VGGNet(VGG-19)을 사용했습니다.

14 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

15 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]

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_15 (Conv2D)           (None, 28, 28, 32)        320
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 28, 28, 64)        18496
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (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 (MaxPooling2 (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.

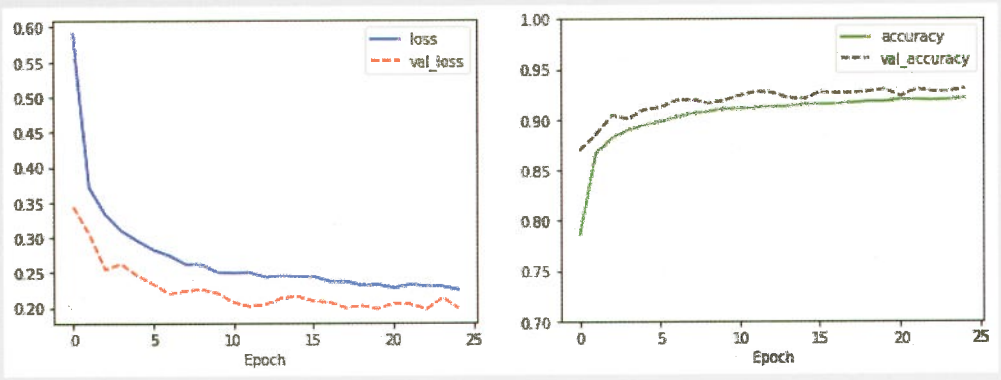
[IN]

```
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.

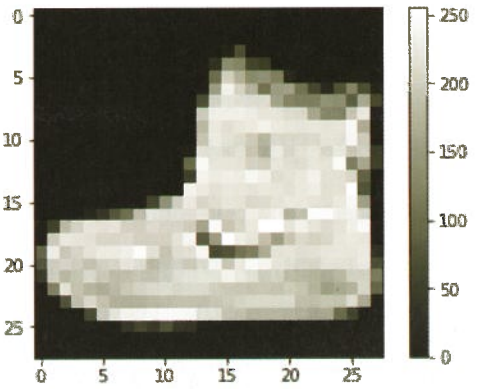


그림 6.16 Fashion MNIST 훈련 데이터의 첫 번째 이미지

tf.keras에는 이러한 이미지 보강 작업을 쉽게 해주는 ImageDataGenerator가 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.tile[16](train_X[0].reshape(28*28),100).reshape(-1,28,28,1),
np.zeros(augment_size), batch_size=augment_size, shuffle=False).next()[0]
```

16 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]

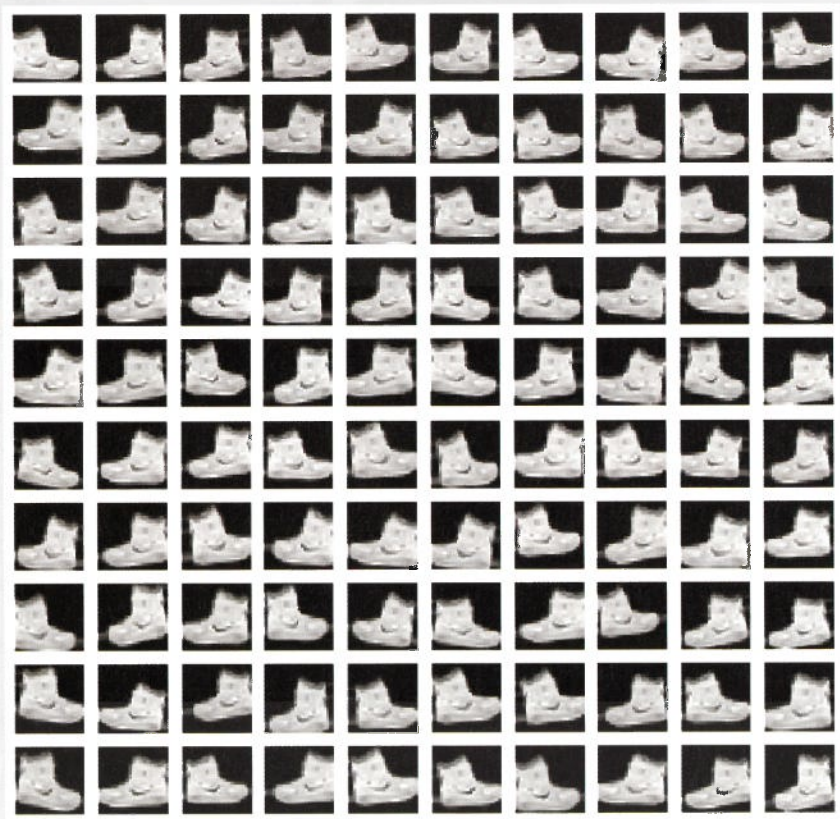


그림 6.16의 신발 이미지가 여러 가지 형태로 조금씩 바뀐 것을 확인할 수 있습니다. 다양한 변형이 가해지기 때문에 각 이미지는 비슷하면서도 서로 조금씩 다릅니다.

ImageDataGenerator의 주요 인수들은 rotation_range, zoom_range, shear_range 등입니다. 가로축으로 이미지를 뒤집는 horizontal_flip은 사용하지만 세로축으로 뒤집는 vertical_flip은 사용하지 않습니다. Fashion MNIST에는 보통 이미지가 위아래로 반듯하게 놓여 있기 때문에 vertical_flip 옵션을 True로 설정하면 대비하지 않아도 될 경우(이미지가 뒤집혀 있는 경우)에 대해서도 대비하게 되어 퍼포먼스가 떨어집니다.

flow() 함수는 실제로 보강된 이미지를 생성합니다. 이 함수는 Iterator라는 객체를 만드는데, 이 객체에서는 값을 순차적으로 꺼낼 수 있습니다. 값을 꺼내는 방법은 next() 함수를 사용하는 것입니다.[17] 한번에 생성할 이미지의 양인 batch_size를 위에서 설정한 augment_size와 같은 100으로 설정했기 때문에 next() 함수로 꺼내는 이미지는 100장이 됩니다. 나머지 부분은 matplotlib.pyplot으로 생성된 보강 이미지를 그래프로 그려주는 부분입니다.

그럼 실제로 훈련 데이터 이미지를 보강하기 위해 다량의 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]
```

17 제가 인터넷에서 찾은 Iterator에 대한 가장 쉬운 설명은 이곳(http://bit.ly/2Y0RUwZ)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 = 30000으로 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 False로 설정하면 됩니다.

randidx는 [2, 25432, 425, …]와 같은 정수로 구성된 넘파이 array이고, 이 array를 이용해 train_X에서 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 ImageDataGenerator의 flow() 함수로 30,000개의 새로운 이미지를 생성합니다.

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

그럼 이제 예제 6.12의 VGGNet 스타일의 네트워크에 ImageDataGenerator로 보강된 훈련 데이터를 학습시켜보겠습니다.

예제 6.16 VGGNet style 네트워크 + 이미지 보강학습

[IN]

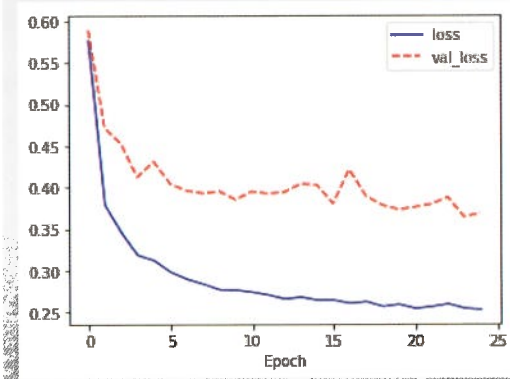
```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(input_shape=(28,28,1), kernel_size=(3,3), filters=32,
padding='same', activation='relu'),
    tf.keras.layers.Conv2D(kernel_size=(3,3), filters=64, padding='same', activation='relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2,2)),
    tf.keras.layers.Dropout(rate=0.5),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

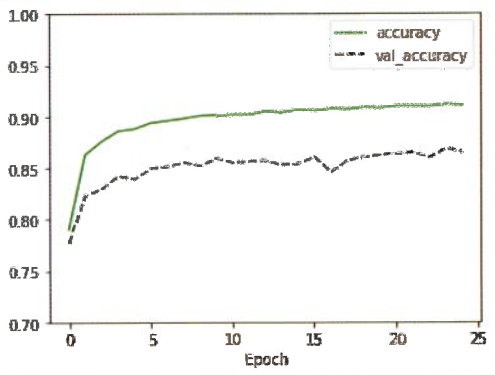
```
    tf.keras.layers.Conv2D(kernel_size=(3,3), filters=256, padding='valid', activation='relu'),
    tf.keras.layers.MaxPool2D(pool_size=(2,2)),
    tf.keras.layers.Dropout(rate=0.5),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=512, activation='relu'),
    tf.keras.layers.Dropout(rate=0.5),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

```
0.7902 - 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)
```






```
[0.2072610476076603, 0.9288]
```

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.