3. 고급 컴퓨터 비전

(이미지에서 특징 감지하기)

이미지에서 특징 감지하기

의류 아이템을 잘 감지하는 신경망을 만들었지만 확실히 제한적입니다. 이 신경망은 아이템 하나가 이미지 중앙에 놓인 작은 흑백 이미지로 훈련되었습니다.

모델을 한 단계 더 발전시키려면 이미지에 있는 특징을 감지해야 합니다. 예를 들어 이미지에 있는 원본 픽셀을 보는 것 대신에 이미지를 어떤 구성 요소로 필터링하는 방법이 있다면 어떨까요? 원본 픽셀 대신 이런 구성 요소를 매칭하면 이미지 안의 내용을 훨씬 더 효과적으로 감지할 수 있습니다. 이전 장에서 보았던 패션 MNIST 데이터셋을 생각해 보죠. 신발을 감지할 때 이미지 아래에 모여 있는 검은 픽셀에 의해 활성화되었을지도 모릅니다.이 부분을 신발이라고 생각했을지 모르죠. 하지만 신발이 중앙에 있지 않거나 이미지 프레임을 채우지 못하면 이런 논리는 성립되지 않습니다.

특징을 감지하는 방법은 우리에게 익숙한 사진과 이미지 처리 방법에서 찾을 수 있습니다. 포토샵이나 김프 같은 도구를 사용해 이미지를 선명하게 만들어봤다면 수학적 필터를 이미지 픽셀에 적용한 것입니다. 이런 필터를 합성곱이라고 합니다. 신경망에 이 필터를 사용해 **합성곱 신경망(convolutional neural network: CNN)**을 만들어 보겠습니다.

3.1 합성곱(convolution)

하나의 함수와 또 다른 함수를 반전 이동한 값을 곱한 다음, 구간에 대해 적분하여 새로운 함수를 구하는 수학 연산자이다.

소리 신호 필터링, 영상 처리 등과 같이 입력과 이에 대한 출력이 존재하는 경우 입력을 목적에 따라 가공해서 원하는 출력을 얻기 위해서 사용하는 연산이다.

"합성곱"이라는 개념을 간단히 정의해 보면 "두 함수 중 하나를 반전(reverse), 이동(shift)시켜가며 나머지 함수와의 곱을 연이어 적분한다"이다.

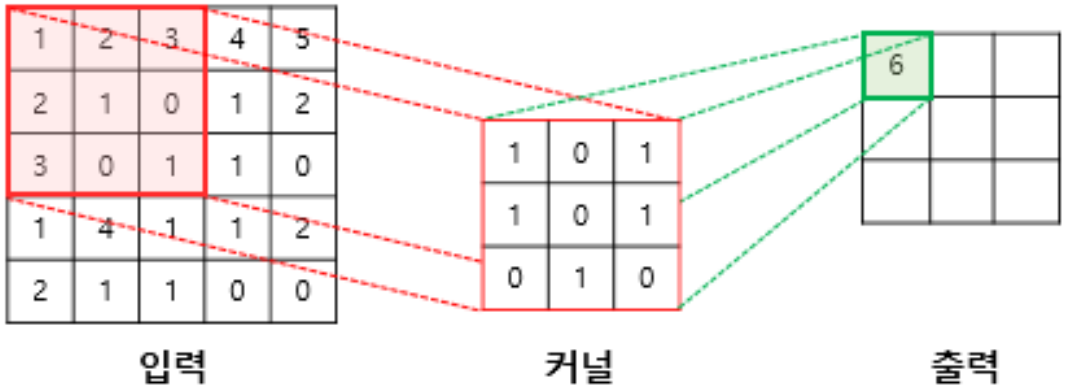
합성곱은 주로 이미지에서 특징을 추출하는 데 사용됩니다. 우선, 작은 필터(커널)를 정의하고 이를 입력 이미지에 적용합니다. 필터는 이미지를 스캔하면서 특정한 패턴이나 특징을 찾아냅니다.

합성곱층은 합성곱 연산을 통해서 이미지의 특징을 추출하는 역할을 합니다. 우선, 합성곱 연산에 대해서 이해해봅시다. 합성곱은 영어로 컨볼루션이라고도 불리는데, 커널(kernel) 또는 필터(filter)라는 n X m 크기의 행렬로 높이(height) X 너비(width) 크기의 이미지를 처음부터 끝까지 겹치며 훑으면서 n X m 크기의 겹쳐지는 부분의 각 이미지와 커널의 원소의 값을 곱해서 모두 더한 값을 출력으로 하는 것을 말합니다. 이때, 이미지의 가장 왼쪽 위부터 가장 오른쪽까지 순차적으로 훑습니다.커널(kernel)은 일반적으로 3 × 3 또는 5 × 5를 사용합니다.

3.1 합성곱(convolution)

예를 통해 이해해봅시다. 아래는 3×3 크기의 커널로 5×5의 이미지 행렬에 합성곱 연산을 수행하는 과정을 보여줍니다. 한 번의 연산을 1 스텝(step)이라고 하였을 때, 합성곱 연산의 네번째 스텝까지 이미지와 식으로 표현해봤습니다.

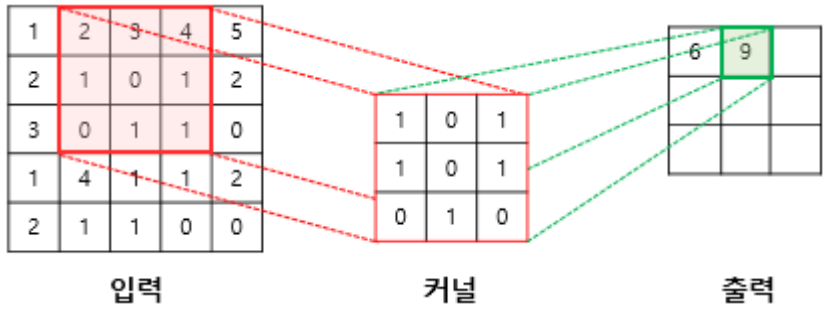
1) 첫번째 스텝



$$(1\times1) + (2\times0) + (3\times1) + (2\times1) + (1\times0) + (0\times1) + (3\times0) + (0\times1) + (1\times0) = 6$$

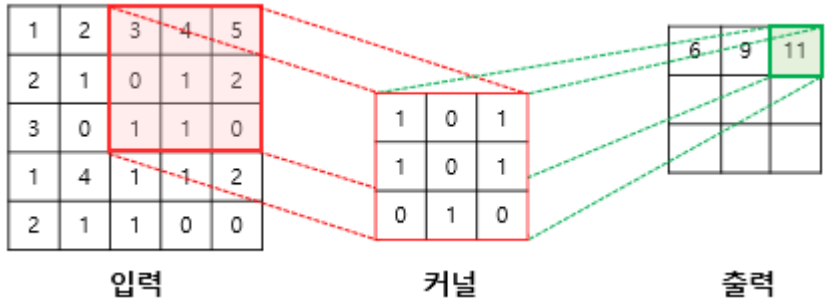
3.1 합성곱(convolution)

2) 두 번째 스텝



(2×1) + (3×0) + (4×1) + (1×1) + (0×0) + (1×1) + (0×0) + (1×1) + (1×0) = 9

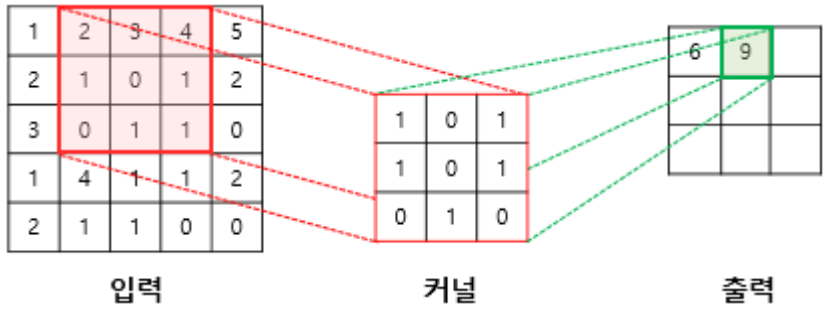
3) 세 번째 스텝



(3×1) + (4×0) + (5×1) + (0×1) + (1×0) + (2×1) + (1×0) + (1×1) + (0×0) = 11

3.1 합성곱(convolution)

4) 네 번째 스텝



(2×1) + (1×0) + (0×1) + (3×1) + (0×0) + (1×1) + (1×0) + (4×1) + (1×0) = 10

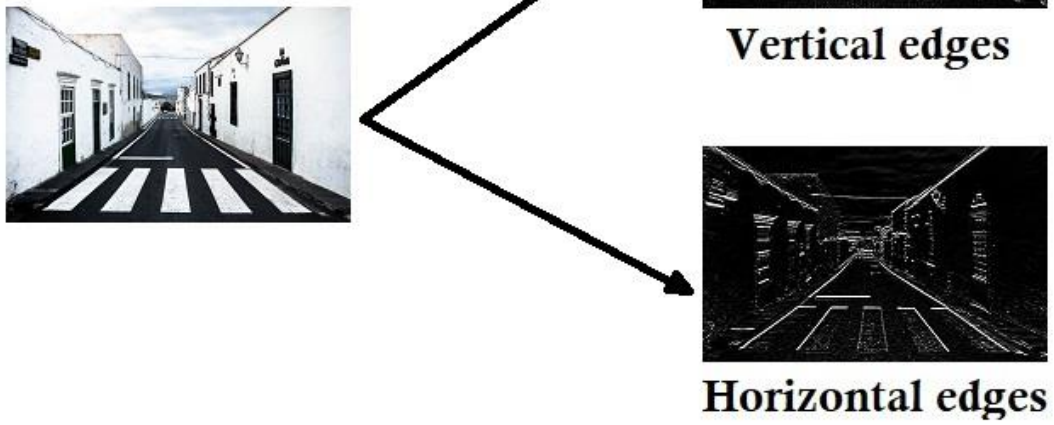
위 연산을 총 9번의 스텝까지 마쳤다고 가정하였을 때, 최종 결과는 아래와 같습니다.

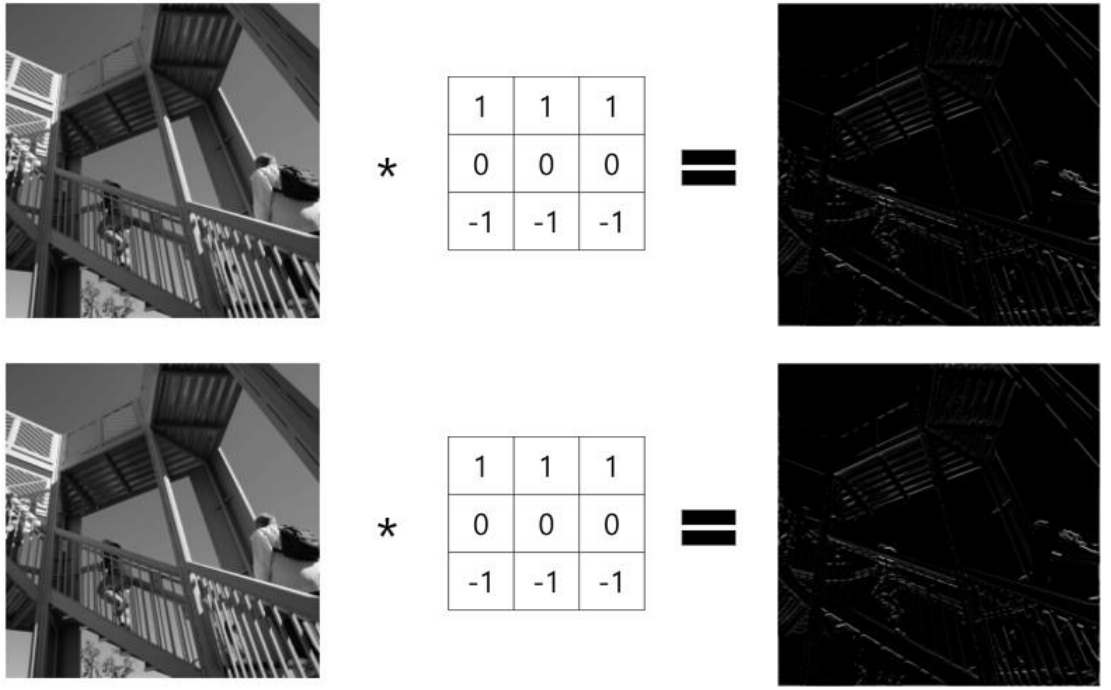
6	9	11
10	4	4
7	7	4

특성 맵(feature map)

3.1 합성곱(convolution)

수직선에 집중하는 커널은 위와 같은 이미지를, 수평선에 집중하는 커널은 아래와 같은 이지미를 출력하게 됩니다. 쉽게 생각해서 수직선을 인식하는 필터는 가운데 열만 1로 채워져 있고, 그 외에는 모두 0입니다. (다른 형태도 있긴 함)따라서 가운데 수직선 부분을 제외하고는 수용장에 있는 모든 것을 무시할 것입니다.혹은 가운데 수직선만 제외하고, 나머지 수직선만 인식할 수 있습니다. (이때는 커널이 가운데만 0, 나머지는 1)이런 식으로 필터는 하나의 특성 맵 (feature map)을 만듭니다.





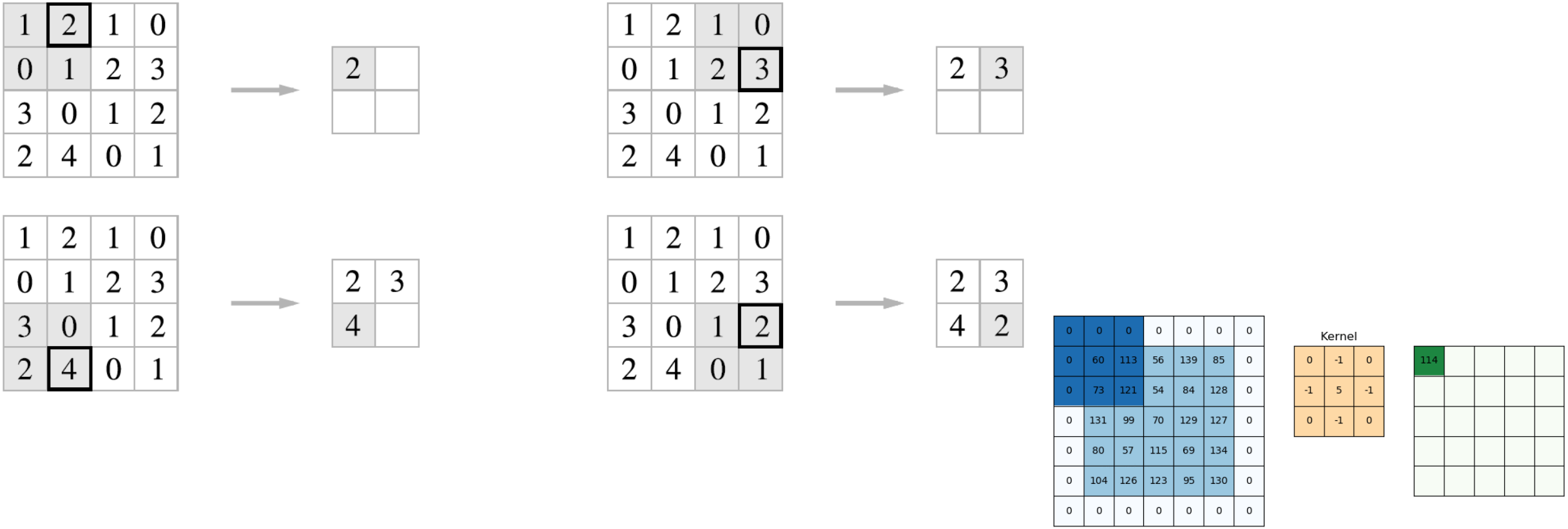
3.2 폴링

풀링(Pooling)은 특성 맵 크기를 줄여 이미지의 요약 정보를 추출하는 기능을 합니다.합성곱과 풀링은 어떤 차이가 있을까요? 합성곱은 필터와 관련이 있습니다. 필터의 패턴과 유사한 영역을 추출합니다. 반면 풀링은 필터가 필요 없습니다. 특정 영역에서 최댓값이나 평균값을 가져와 요약 정보를 구하기 때문입니다.요약 정보를 구하는 까닭, 곧 풀링의 목적은 무엇일까요? 첫 번째는 특성 맵 크기를 줄여 연산 속도를 빠르게 하기 위함입니다. 두 번째는 이미지에서 물체의 위치가 바뀌어도 같은 물체로 인식하기 위해서입니다. 고양이 이미지를 떠올려봅시다. 이미지에서 고양이가 가운데 있든 왼쪽에 있든 똑같은 고양이입니다. 물체 위치가 달라져도 같은 물체죠. 합성곱 연산은 물체 위치가 바뀌면 같은 물체로 인식하지 못합니다. 이 문제를 해결하기 위해 풀링이 필요합니다. 풀링은 특정 영역의 요약 정보(대푯값)를 가져오므로 위치가 약간 변해도 같은 물체라고 판단합니다. 이런 성질을 위치 불변성(Location Invariance)이라고 합니다.

일반적으로 합성곱 층(합성곱 연산 + 활성화 함수) 다음에는 풀링 층을 추가하는 것이 일반적입니다. 풀링 층에서는 특성 맵을 다운샘플링하여 특성 맵의 크기를 줄이는 풀링 연산이 이루어집니다. 풀링 연산에는 일반적으로 최대 풀링(max pooling)과 평균 풀링(average pooling)이 사용됩니다.

3.2 폴링

풀링에는 대표적으로 최대 풀링과 평균 풀링이 있습니다. 최대 풀링(Max Pooling)은 풀링 영역에서 가장 큰 값을 취하는 방법이고, 평균 풀링(Average Pooling)은 풀링 영역의 평균값을 구하는 방법입니다. 최대 풀링은 특정 영역에서 가장 뚜렷한(밝은) 부분을 추출하고, 평균 풀링은 특정 영역의 평균적인 특징을 추출합니다. 다음 그림은 풀링 크기를 2 x 2로 설정하여 최대 풀링을 하는 예입니다.



3.3 합성곱 신경망 만들기

전체 코드

```
import tensorflow as tf

data = tf.keras.datasets.fashion_mnist
(training_images, training_labels), (test_images, test_labels) = data.load_data()
training_images = training_images.reshape(60000, 28, 28, 1)
training_images = training_images / 255.0
test_images = test_images.reshape(10000, 28, 28, 1)
test_images = test_images / 255.0
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

3.3 합성곱 신경망 만들기

```
model.summary()

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


model.fit(training_images, training_labels, epochs=50)

model.evaluate(test_images, test_labels)

classifications = model.predict(test_images)
print(classifications[0])
print(test_labels[0])
```

3.3 합성곱 신경망 만들기

2장에서 패션 MNIST 이미지를 인식하는 신경망을 만들었습니다. 이를 합성곱 신경망으로 바꾸려면 간단하게 모델 정의에 합성곱 층을 쓰면 됩니다. 그리고 폴링 층도 추가하겠습니다.
합성곱 층으로 tf.keras.layers.Conv2D를 사용합니다. 이 클래스는 층에 사용할 합성곱 필터 개수, 필터 크기, 활성화 함수 등을 매개변수로 받습니다.
예를 들어 신경망의 첫 번째 층으로 합성곱을 사용하는 예는 다음과 같습니다.

```
tf.keras.layers.Conv2D(64, (3, 3), activation='relu', input_shape=(28, 28, 1)),
```

여기에서는 64개의 합성곱 필터를 학습합니다. 필터는 랜덤하게 초기화되고 시간이 지남에 따라 입력을 레이블로 매핑하기 위해 가장 좋은 필터 값을 학습합니다. (3, 3)은 필터의 크기입니다. 앞에서 3 X 3 필터를 언급했는데 바로 여기서 크기를 지정합니다. 이 크기가 가장 일반적이며 다른 값으로 바꿀 수 있지만 일반적으로 5 X 5와 7 X 7 같은 홀수를 사용합니다.
activatio과 input_shape은 이전과 동일합니다. 여기서는 패션 MNIST를 사용하기 때문에 이미지 크기는 그대로 28 X 28입니다. 하지만 Conv2D 층은 컬러 이미지를 위해 설계되었기 때문에 세 번째 차원을 1로 지정해야 합니다. 따라서 입력 크기는 28 X 28 X 1이 됩니다.
컬러 이미지는 일반적으로 R, G, B 값으로 저장되므로 컬러 이미지의 세번째 차원은 3입니다.
산경망에서 폴링 층은 다음처럼 사용합니다. 보통 합성곱 층 다로 뒤에 적용합니다.

```
tf.keras.layers.MaxPooling2D((2, 2)),
```

3.3 합성곱 신경망 만들기

입력 크기가 Conv2D 층이 기대한 것과 일치해야 하기 때문에 28 X 28 X 1로 업데이트한 것을 기억하시나요? 데이터도 이에 맞게 크기를 바꾸어야 합니다. 28 X 28은 이미지에 있는 픽셀의 개수이고 1은 컬러 채널의 개수입니다. 일반적으로 흑백 이미지는 채널이 1개이고 컬러 이미지는 R, G, B를 위한 3개의 채널을 가집니다. 각 채널의 값은 해당 색상의 강도를 나타냅니다.
따라서 이미지를 정규화하기 전에 배열에 차원을 추가합니다. 다음 코드는 훈련 세트에 있는 6만 개의 이미지를 28 X 28 크기에서 28 X 28 X 1 크기로 바꿉니다.

```
training_images = training_images.reshape(60000, 28, 28, 1)
```

테스트 데이터셋도 동일하게 바꿉니다.
심층 신경망(deep neural network: DNN)을 사용했을 때 입력을 첫 번째 Dense 층에 주입하기 전에 Flatten 층에 먼저 통과시켰습니다.
여기에서는 Flatten 층이 없고 대신 입력 크기를 지정합니다. 합성곱과 폴링 층 다음에는 Flatten 층으로 데이터를 펼쳐서 Dense 층에 전달합니다.

2장에서 했던 것처럼 동일한 데이터로 이 신경망을 50번의 에폭 동안 훈련하면 정확도가 높아지는 것을 볼 수 있습니다. 이전에는 50번의 에폭 동안 훈련해 테스트 세트에서 89% 정확도를 달성했지만 이번에는 절반 정도인 24나 25번 에폭에서 99% 정확도를 달성합니다. 신경망에 합성곱 층을 추가해 이미지를 분류하는 능력을 크게 향상시켰습니다.

3.4 합성곱 신경망 살펴보기

```
model.summary()
```

model.summary 메서드로 모델을 분석할 수 있습니다. 먼저 Output Shape 열을 살펴보죠. 첫 번째 층은 28 X 28 크기의 이미지를 받아 64개의 필터를 적용합니다. 하지만 필터 크기가 3 X 3 이기 때문에 이미지 경계에 있는 1픽셀을 놓치게 되고 합성곱 출력 크기가 26 X 26 픽셀로 줄어들게 됩니다. 이미지의 각 픽셀에 필터를 놓았을 때 시작할 수 있는 첫 번째 필터의 중심은 두 번째 행의 두 번째 열입니다. 동일한 현상이 오른쪽 끝과 맨 아래에도 일어납니다. 따라서 크기가 A X B 픽셀인 이미지에 3 X 3 필터를 적용하면 출력은 (A - 2) X (B -2) 크기가 됩니다. 마찬가지로 5 X 5 필터를 사용하면 (A - 4) X (B - 4)가 되는 식입니다. 예제에서는 28 X 28 이미지와 3 X 3 필터를 사용했기 때문에 출력 크기는 26 X 26입니다.폴링 크기는 2 X 2이므로 이미지 크기는 각 축 방향으로 절반씩 줄어들어 13 X 13이 됩니다. 다음 합성곱 층은 이를 11 X 11로 줄입니다. 그 다음 폴링에서 오른쪽과 아래쪽 가장자리 픽셀이 버려지고 이미지는 5 X 5가 됩니다.이미지가 두 개의 합성곱 층을 통과하면 결과적으로 많은 5 X 5의 이미지가 만들어집니다. 파라미터가 얼마나 많을까요? Param# 열에서 이를 확인할 수 있습니다.

각 합성곱은 3 X 3 필터 절편을 가집니다. 밀집 층을 사용했을 때 각 층은 Y = mX +c의 같습니다. 여기에서 m이 파라미터(가중치)고 c는 절편입니다. 필터가 3 X 3이기 문에 9개의 파라미터가 있다는 것을 제외하면 이와 매우 비슷합니다. 합성곱 필터가 64개이므로 전체 파라미터는 640개 입니다(파라키터 9개와 절편을 더하면 총 10개가 됩니다. 여기에 필터 개수 64를 곱합니다.)

3.4 합성곱 신경망 살펴보기

다음 합성곱 층은 크기가 9인 64개의 필터를 가지며 각 필터는 이전의 64개 필터와 곱해집니다. 새로운 64개 필터마다 절편이 있으므로 전해 파라미터 개수는 (64 X (64 X 9)) + 64입니다. 즉 두 번째 합성곱 층이 학습할 파라미터는 36,928개입니다.
조금 헷갈린다면 첫 번째 층의 합성곱 필터 개수를 10개로 바꾸어보죠. 그러면 두 번째 층의 파라미터는 5,824개가 됩니다((64 X (10 X 9)) + 64).
두 번째 층을 통과하면 이미지는 크기는 5 X 5가 되고 64개의 채널을 가집니다. 이를 모두 곱하면 1,600개입니다. 이를 펼쳐서 128개 뉴런을 가진 밀집 층에 주입합니다. 128개의 뉴런은 각각 가중치와 절편을 가지므로 밀집 층이 학습할 파라미터 개수는 ((5 X 5 X 64) X128) + 128인 204,928개입니다.

10개의 뉴런으로 구성된 마지막 밀집 층은 이전 층의 128개 뉴런에서 나오는 출력을 받습니다. 따라서 마지막 층의 파라미터는 (128 X 10) + 10인 1,290개가 됩니다.

전체 파라미터 개수는 이를 모두 더한 243,786개입니다.

신경망을 훈련하는 것은 입력 이미지를 레이블에 매칭하기 위해 이런 243,786개 파라미터의 최상의 조합을 학습하는 것입니다. 파라미터 많아 학습이 오래 걸리지만 결과에서 확인할 수 있듯이 더 정확한 모델을 만듭니다!
물론 이 데이터셋은 28 X 28 크기의 흑백 이미지이고 중앙에 정렬되어 있습니다.

3.4 말과 사람을 구별하는 CNN 만들기

이전에 배웠던 합성곱과 합성곱 신경망을 확쟁해 특징의 위치가 동일하지 않은 이미지를 분류해보겠습니다.
이를 위해 말과 사람 데이터셋을 만들었습니다.

3.5.1 말-사람 데이터셋

이번 절에서 사용하는 데이터셋은 300 X 300픽셀 이미지 여러 개를 담고 있으며 여러 포즈의 말과 사람이 거의 절반씩 구성되어 있습니다.

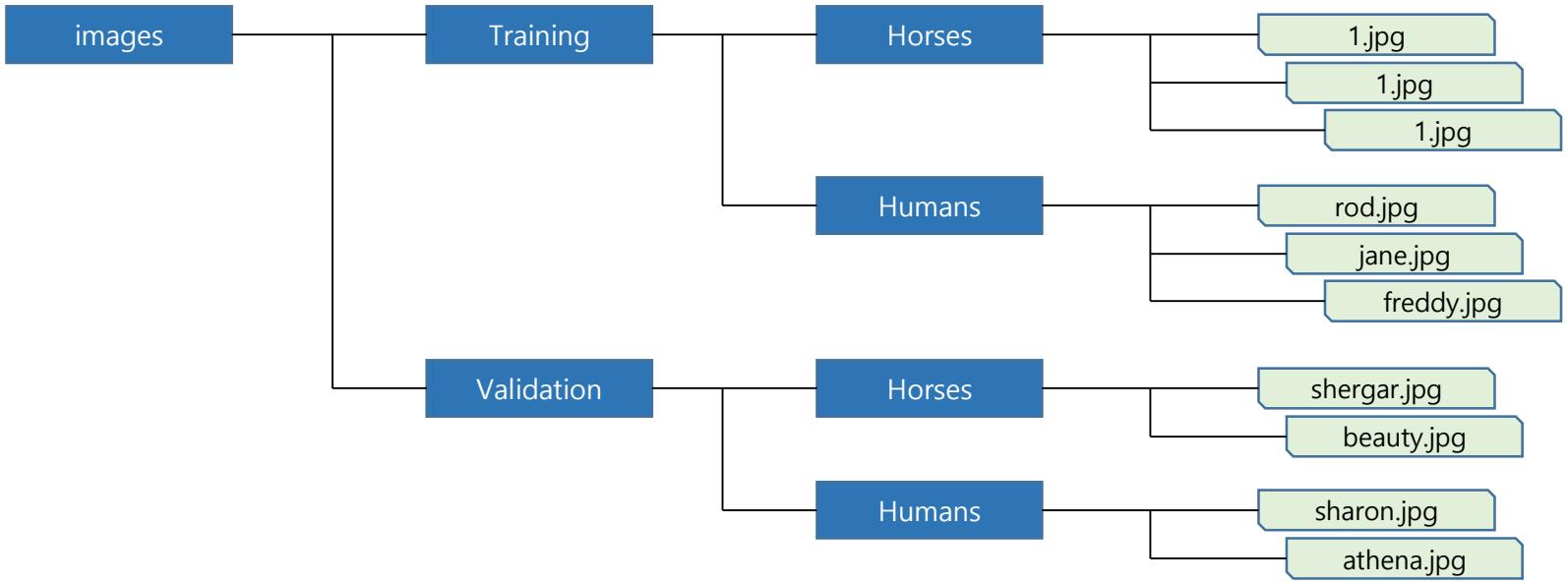



피사체의 방향과 자세가 다르고 이미지 구성이 제각각입니다. 위의 말 두 마리만 보아도 머리 방향이 다릅니다. 한 이미지는 축소되어 전체 모습이 보이고 한 이미지는 확대되어 머리와 몸의 일부만 보입니다. 마찬가지로 두 사람의 이미지도 조명, 피부색, 자세가 다릅니다. 남자는 허리에 손을 얹고 있고 여자는 손을 펴고 있습니다. 이미지에는 나무와 해변 같은 배경이 포함되어 있어 분류 모델이 배경에 영향을 받지 않고 말과 사람을 결정할 수 있는 중요한 특징을 찾아야 합니다.
Y 2X -1을 예측하는 예제나 흑백 의류 이미지를 분류하는 예제는 전통적인 코딩으로 해결할 수도 있지만 이 예제는 확실히 훨씬 더 어렵고 머신러닝을 사용해 문제를 해결해야만 하는 영역입니다.

3.5.2 케라스의 ImageDataGenerator

지금까지 사용했던 패션 MNIST 데이터셋은 레이블을 함께 제공합니다. 훈련 세트와 테스트 세트 레이블이 별도의 파일에 저장되어 있었습니다. 하지만 이미지 기반 데이터셋 상당수는 이런 식으로 파일을 제공하지 않는 경우가 많습니다. 말-사람 데이터셋도 마찬가지입니다. 레이블 대신 이미지가 디렉토리에 종류별로 나뉘어져 있습니다. 케라스의 ImageDataGenerator를 사용하면 이런 디렉토리 구조를 사용해 자동으로 이미지에 레이블을 할당할 수 있습니다.

ImageDataGenerator를 사용하려면 레이블 이름으로 된 디렉토리를 구성해야 합니다. 예를 들어 말-사람 데이터셋은 1,000개 이상의 이미지로 구성된 훈련 데이터 ZIP파일과 256개 이미지로 구성된 검증 데이터를 위한 또 다른 zip파일을 제공합니다. gnsfus 데이터와 검증 데이터를 다운로드해 로컬 디렉토리에 압축을 풀 때 아래의 그림과 같은 디렉토리 구조로 만들어야 합니다.



3.5.2 케라스의 ImageDataGenerator

다음 코드는 훈련 데이터를 다운로드해 앞의 디렉토리 구조로 압축을 푸는 코드입니다.

```
import urllib.request
import zipfile

url = "https://storage.googleapis.com/learning-datasets/horse-or-human.zip"
file_name = "horse-or-human.zip"
training_dir = 'horse-or-human/training/'
urllib.request.urlretrieve(url, file_name)

zip_ref = zipfile.ZipFile(file_name, 'r')
zip_ref.extractall(training_dir)
zip_ref.close()
```

위 코드는 훈련 데이터 zip 파일을 다운로드 해 horse-or-human/training 디렉토리에 압축을 풉니다. 이 디렉토리는 말과 사람 이미지를 담은 서브 디렉토리를 포함합니다.

3.5.2 케라스의 ImageDataGenerator

ImageDataGenerator를 사용하려면 다음 코드를 실행합니다.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# 전체 이미지를 1./255로 스케일을 조정합니다.
train_datagen = ImageDataGenerator(rescale=1/255)

train_generator = train_datagen.flow_from_directory(
    training_dir,
    target_size=(300, 300),
    class_mode='binary'
)
```

먼저 ImageDataGenerator 객체 train_datagen을 만듭니다. 그 다음 훈련 과정 동안 디렉토리를 순회하면서 이미지를 생성하는 반복자 객체 train_generator를 만듭니다. 대상 디렉토리는 braining_dir입니다. 이미지 크기를 지정하는 target_size와 레이블 종류를 지정하는 class_mode 같은 매개변수를 지정합니다. class_mode는 여기에서처럼 이미지가 두 개인 경우 binar이고 두 개 이상인 경우에는 categorical로 지정합니다.

3.5.3 말-사람 데이터셋을 위한 CNN 구조

이 데이터셋에 패션 MNIST 데이터셋 사이에는 중요한 차이가 있어 이미지 분류 신경망 구조를 설계할 때 다음 세 가지를 유념해야 합니다. 첫째, 이미지가 300 X 300 픽셀로 훨씬 크기 때문에 많은 층이 필요합니다. 둘째, 흑백이 아니라 컬러 이미지라서 채널이 하나가 아니라 세 개입니다. 셋째, 두 종류의 이미지만 있으므로 하나의 출력 뉴런을 사용하는 이진 분류기를 만들 수 있습니다. 이 뉴런은 한 클래스일 때는 0을 또 다른 클래스일 때는 1에 가까운 값을 출력합니다. 이런 점을 유념하면서 신경망 구조를 만들어 보겠습니다.

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(16, (3,3), activation='relu',  input_shape=(300, 300, 3)),
    tf.keras.layers.MaxPooling2D((2, 2)),      tf.keras.layers.Conv2D(32, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),      tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),      tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),      tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),      tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

3.5.3 말-사람 데이터셋을 위한 CNN 구조

우선 맨 첫 번째 층입니다. 3 X 3 크기의 필터 16개를 사용하고 입력 이미지 크기는 (300, 300, 3)입니다. 입력 이미지 크기가 300 X 300이고 컬러 이미지이므로 채널이 세 개입니다. 이전에 사용했던 패션 MNIST 데이터셋은 흑백이라 채널이 하나였습니다.
마지막 출력 층에는 하나의 뉴런만 있습니다. 이진 분류기를 만들 때 하나의 뉴런 출력을 시그모이드 함수로 활성화하면 이진 분류를 얻을 수 있기 때문입니다. 시그모이드 함수의 목적은 이진 분류에 적합하도록 샘플 한 세트를 0으로 다른 샘플 세트 하나를 1로 유도하는 것입니다.
그 다음 여러 개의 합성곱 층을 쌓았습니다. 입력 이미지가 꽤 크기 때문에 층을 거듭하면서 특징이 강조된 작은 이미지를 많이 만들기 위해서입니다.

```
model.summary()
```

데이터가 합성곱 층과 폴링 층을 모두 통과하면 7 X 7 크기가 됩니다. 이론적으로는 활성화된 특성 맵이라고 부르며 비교적 간단한 49개의 픽셀로 구성됩니다. 이 특성 맵을 밀집 층에 전달해 적절한 레이블에 매핑시킬 수가 있습니다.
물론 이전 신경망보다 파라미터 개수가 훨씬 많아졌고 훈련 시간도 더 길어졌습니다. 이 신경망은 파라미터 170만 개를 학습할 것입니다.

3.5.3 말-사람 데이터셋을 위한 CNN 구조

이 신경망을 훈련하기 위해 손실 함수와 옵티마이저로 컴파일합니다. 이 경우 손실 함수는 이진 클로스 엔트로피를 사용할 수 있습니다. 이름에서 알 수 있듯이 클래스가 두 개인 경우 사용하는 손실 함수입니다. 새로운 옵티마이저인 RMSprop에 학습 속도를 제어하는 학습률 매개변수를 지정해 보겠습니다. 코드는 다음과 같습니다.

```
from tensorflow.keras.optimizers import RMSprop

model.compile(loss='binary_crossentropy',
              optimizer=RMSprop(learning_rate=0.001),
              metrics=['accuracy'])
```

fit 메소드에 앞에서 만든 training_generator를 전달해 모델을 훈련합니다.

```
model.fit(train_generator, epochs=15
```

3.5.3 말-사람 데이터셋을 위한 CNN 구조

이 예제는 콜랩으로 실행할 수 있습니다. 하지만 로컬 컴퓨터에서 실행하려면 pip install pillow 명령을 사용해서 Pillow를 설치해야 합니다.
텐서플로 케라스에서 model.fit 메소드를 사용해 훈련 데이터와 레이블을 매핑하도록 훈련합니다. 텐서플로 이전 버전에서 데이터 제너레이터를 사용할 때는 model.fit_generato를 사용했습니다.
최신 텐서플로 버전에서는 둘 다 사용할 수 있습니다.
이 신경망은 15번의 에폭만에 훈련 세트에서 95% 이상의 매우 인상적인 정확도를 달성합니다. 물론 훈련 데이터의 성능일 뿐이므로 신경망이 본 적 없는 데이터에 대한 성능을 의미하는 것은 아닙니다.
제너레이터를 사용해 검증 세트를 추가하고 성능을 측정해 실제로 이 모델이 얼마나 잘 작동할지 가늠해보겠습니다.

3.5.3 말-사람 데이터셋을 위한 CNN 구조

최신 텐서플로에서는 ImageDataGenerator 대신 image_dataset.from_directory 함수를 사용하는 것이 좋습니다. 이 함수는 tf.data.Datasets 객체를 반환합니다. 이 함수에서는 이미지 스케일 조정과 같은 전처리 기능이 포함되어 있지 않습니다. 대신 이런 전처리 기능은 모델 층의 일부로 추가해야 합니다. 이를 위해 캐러스는 다양한 전처리 층을 제공합니다(https://keras.io/api/layers/preprocessing_layers).
앞의 예제를 image_dataset_from_directory 함수와 Rescaling 층을 사용하면 다음과 같이 변경할 수 있습니다.

```
train_ds = tf.keras.utils.image_dataset_from_directory(
    training_dir,    image_size=(300, 300),    label_mode='binary'
)
model = tf.keras.models.Sequential([
    tf.keras.layers.Rescaling(1./255, input_shape=(300, 300, 3)),
    tf.keras.layers.Conv2D(16, (3,3), activation='relu'),      tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu'),     tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),     tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),     tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),     tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),      tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

3.5.3 말-사람 데이터셋을 위한 CNN 구조

```
model.compile(loss='binary_crossentropy',
              optimizer=RMSprop(learning_rate=0.001),   metrics=['accuracy'])

model.fit(train_ds, epochs=15)
```

3.5.4 검증 세트 추가하기

모델을 검증하려면 훈련 데이터셋과 별개로 검증 데이터셋이 필요합니다. 전체 데이터셋에서 직접 검증 세트를 덜어내야 하는 경우도 있지만 말-사람 데이터셋의 경우에는 별도의 검증 세트를 다운로드할 수 있습니다. 훈련 데이터를 다운로드할 때 사용했던 것과 유사한 코드로 검증 세트를 다운로드하고 별도의 디렉토리에 압축을 풉니다.

```
validation_url = "https://storage.googleapis.com/learning-datasets/validation-horse-or-human.zip"

validation_file_name = "validation-horse-or-human.zip"
validation_dir = 'horse-or-human/validation/'
urllib.request.urlretrieve(validation_url, validation_file_name)

zip_ref = zipfile.ZipFile(validation_file_name, 'r')
zip_ref.extractall(validation_dir)
zip_ref.close()
```

3.5.4 검증 세트 추가하기

검증 데이터가 준비되면 이미지를 로드하기 위해 또 다른 ImageDataGenerator 객체를 만듭니다.

```
validation_datagen = ImageDataGenerator(rescale=1/255)

validation_generator = validation_datagen.flow_from_directory(
    validation_dir,
    target_size=(300, 300),
    class_mode='binary'
)
```

텐서플로에서 검증 데이터를 적용하려면 model.fit 메소드에 에폭마다 모델을 테스트하기 위해 사용할 검증 데이터를 저장하면 됩니다. 이를 위해 validation_data 매개변수에 앞에서 만든 검증 제너레이터를 전달합니다.

```
model.fit(train_generator,
          epochs=15,
          validation_data=validation_generator)
```

3.5.4 검증 세트 추가하기

15번의 에폭 후에 훈련 세트에서 99% 이상의 정확도를 달성하였지만 검증 세트에서는 약 88%의 정확도를 달성합니다. 이는 이전 장에서 본 것처럼 모델이 과대적합되었다는 징후입니다.

훈련에 사용한 이미지 개수가 적도 다양하다는 것을 생각하면 성능이 나쁜 편은 아닙니다. 데이터 부족에 당면하기 시작했지만 모델 성능을 향상할 수 있는 몇 가지 기법이 있습니다.

3.5.5 말 또는 사람 이미지로 테스트하기

모델을 구출할 수 있어 좋지만 직접 사용해보고 싶을 것입니다. 콜랩을 사용하면 모델을 테스트하기가 아주 쉽습니다. 다음 코들르 사용하면 콜랩에서 로컬 컴퓨터에 있는 이미지를 업로드할 수 있습니다. 파일을 업로드하지 않는 경우에는 hh_image_1.jpg, hh_image_2.jpg, hh_image_3.jpg가 있다고 가정합니다.

```
import sys

# 코랩을 사용중인지 확인합니다.
if 'google.colab' in sys.modules:
    from google.colab import files
    uploaded = files.upload()
    sample_images = ['/content/' + fn for fn in uploaded.keys()]

    # 업로드된 파일이 없으면 깃허브에서 다운로드합니다.
    if len(uploaded) < 1:
        import gdown
        base_url = 'https://github.com/rickiepark/aiml4coders/raw/main/ch03/'
        for i in range(1,4):
            gdown.download(base_url + 'hh_image_{}.jpg'.format(i))
        sample_images = ['/content/hh_image_{}.jpg'.format(i) for i in range(1,4)]
```

3.5.5 말 또는 사람 이미지로 테스트하기

```
# 로컬 컴퓨터면 ch03 폴더에 있는 이미지를 사용합니다.
else:
    sample_images = ['hh_image_{}.jpg'.format(i) for i in range(1,4)]
```

3.5.5 말 또는 사람 이미지로 테스트하기

그 다음 훈련한 모델을 사용해 테스트 이미지를 분류해 보겠습니다.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from keras.preprocessing import image

for fn in sample_images:
    # 이미지 출력
    plt.imshow(mpimg.imread(fn))
    plt.show()

    # 이미지 불러오기
    img = tf.keras.utils.load_img(fn, target_size=(300, 300))
    x = tf.keras.utils.img_to_array(img)
    x = np.expand_dims(x, axis=0)

    classes = model.predict(x)
```

3.5.5 말 또는 사람 이미지로 테스트하기

```
print('모델 출력:', classes[0][0])
if classes[0][0] > 0.5:
    print(fn + "는 사람입니다.")
else:
    print(fn + "는 말입니다.")
print('--------------------')
```

여러 개의 이미지를 업로드해 모델의 예측을 확인할 수 있습니다.

3.5.5 말 또는 사람 이미지로 테스트하기

```
# 이미지 불러오기
img = tf.keras.utils.load_img(fn, target_size=(300, 300))
x = tf.keras.utils.img_to_array(img)
x = np.expand_dims(x, axis=0)
```

이 코드는 pat 경로에 있는 이미지를 로드합니다. target_size를 300 X 300으로 지정했습니다. 임의의 크기의 이미지를 업로드할 수 있지만 모델에 주입하려면 반드시 300 X 300 크기여야 합니다. 모델이 이 크기의 이미지를 인식하도록 훈련되었기 때문입니다. 따라서 첫 번째 줄은 이미지를 로드한 다음 300 X 300 크기로 변경합니다.

그 다음 코드는 이미지를 2D 배열로 변환합니다. 하지만 모델을 만들 때 input_shape 매개변수에 지정했듯이 모델은 3D 배열을 기대합니다. 넘파이 expand_dims 함수를 사용해 배열에 새로운 차원을 추가하면 이 문제를 쉽게 해결할 수 있습니다.

이제 이미지를 적절한 포맷으로 변환했으므로 다음 코드에서 분류 작업을 수행할 수 있습니다.

```
classes = model.predict(x)
```

3.5.5 말 또는 사람 이미지로 테스트하기

모델은 분류 결과를 담은 배열을 반환합니다. 이 경우 출력 결과가 하나의 숫자이고 classes는 2차원 배열입니다. 첫 번째 차원을 따라 샘플이 놓여 있고 두 번째 차원을 차원을 따라 각 클래스에 대한 점수가 나열됩니다. 이 예제는 이진 분류이므로 양성 클래스(사람)에 대한 점수 하나만 담겨 있습니다. 이 배열의 첫 번째 원소의 값을 조사하면 레이블을 구분할 수 있습니다. 0.5보다 크면 사람입니다.

```
if classes[0][0] > 0.5:
    print(fn + "는 사람입니다.")
else:
    print(fn + "는 말입니다.")
```

하지만 데이터셋에는 근본적인 문제가 있습니다. 훈련 세트가 모델이 실전에서 만날 수 있는 가능한 상황을 모두 나타내지는 못하기 때문에 모델은 항상 훈련 세트에 어느 정도 과도하게 전문화됩니다.
결과적으로 사람 사진이 말로 분류할 가능성이 있고 실제로도 그런 현상이 나타납니다.
그렇다면 해결책은 무엇일까요? 확실한 것은 기존에 없던 특정 자세의 사람이 담긴 사진을 훈련 데이터에 더 추가하는 것입니다. 하지만 이 방법이 항상 가능하지는 않습니다. 다행이 테서플로에는 가상으로 데이터셋을 확장할 수 있는 멋진 기능이 있습니다. 이를 이미지 증식이라고 합니다.

3.6 이미지 증식

이전 절에서는 비교적 작은 데이터셋에서 훈련해 말-사람 분류 모델을 만들었습니다. 그 결과 이전에 본 적 없는 이미지를 분류할 때 문제에 부딪혔습니다. 훈련 세트에 포함되지 않은 자세를 취한 여성을 말이라고 분류했습니다.
이런 문제를 해결하는 한 가지 방법은 이미지 증식입니다. 이미지 증식은 텐서플로가 데이터를 로드할 때 이미지에 여러 변환을 적용해 새로운 데이터를 추가로 만드는 기법입니다.

다음과 같이 여러 가지 다양한 변환을 적용해 훈련 세트를 늘릴 수 있습니다.

- 회전
- 수평이동
- 수직이동
- 기울임
- 확대
- 반전

3.6 이미지 증식

다음과 같이 ImageDataGenerator로 이미지를 로드했기 때문에 한 가지 변환인 정규화를 적용했습니다.

```
train_datagen = ImageDataGenerator(rescale=1/255)
```

다른 변환은 ImageDataGenerator를 사용하면 쉽게 적용할 수 있습니다. 예를 들면 다음과 같습니다.

```
train_datagen = ImageDataGenerator(
        rescale=1/255,
        rotation_range = 40,                #왼쪽 또는 오른쪽으로 40도까지 랜덤하게 기울입니다.
        width_shift_range=0.2,              #수직 또는 수평으로 20%까지 랜덤하게 이동합니다.
        height_shift_range=0.2,
        shear_range=0.2,                    #20%까지 랜덤하게 기울입니다.
        zoom_range=0.32,                    #20%까지 랜덤하게 확대합니다.
        horizontal_flip=True,               #수평 또는 수직으로 랜덤하게 뒤집습니다.
        fill_mode='nearest'                 #이동하거나 기울인 후에 누락된 픽셀을 근처 픽셀로 채웁니다.
)
```

3.6 이미지 증식

이런 매개변수로 훈련 제너레이터를 다시 만들어 훈련하면 이미지 처리 때문에 훈련 시간이 더 오래 걸리는 것을 알 수 있습니다. 또한 모델 정확도가 이전만큰 높지 않을 수 있습니다. 이전에는 대부분 균일한 데이터셋에 과대적합되었기 때문입니다.
데이터 증식 기법을 사용해 15번의 에폭 동안 훈련하니 정확도가 조금 낮아졌습니다. 하지만 검증 정확도는 더 높아졌습니다(검증 정확도가 훈련 정확도보다 높거나 비슷하면 모델이 과소 적합되었다는 것이므로 매개변수를 조정하는 것이 좋습니다).
이전에 잘못 분류했던 이미지는 이번에 어떤 결과가 나올까요? 이제는 모델이 올바르게 예측합니다. 이미지 증식 덕분에 모델이 이런 이미지도 사람이라는 것을 학습할 수 있는 훈련 세트가 충분히 갖춰졌다는 의미입니다. 하나의 샘플 데이터에 불과하므로 실제 데이터의 결과를 대표하지 못할 수 있지만 올바른 방향으로 한 걸음 나아가는 과정입니다

3.7 전이 학습

모델의 성능을 높일 수 있는 또 다른 기법은 다른 곳에서 이미 학습한 특성을 사용하는 것입니다. 많은 연구자들이 수천 개의 클래스를 가진 대용량 데이터셋(수백만 개의 이미지)에서 훈련한 대규모 모델을 공개합니다. 전이 학습(transfer learning)이란 개념을 사용하면 이런 모델이 학습한 특성을 여러분만의 데이터에 적용할 수 있습니다.
전이 학습을 하면 모델을 더 개선할 수 있습니다. 전이 학습의 아이디어는 간단합니다. 데이터셋의 밑바닥부터 합성곱 필터를 학습하기보다 더 많은 특성을 가진 대규모 데이터셋에서 학습된 필터를 사용하면 어떨까요? 미리 학습된 필터를 사용해 말-사람 데이터로 모델을 훈련할 수 있습니다.
모델이 훈련되고 나면 모델 구조(층의 필터 개수, 필터 크기 등)와 함께 층은 필터 값, 가중치 절편을 나타내는 일련의 숫자에 불과합니다. 따라서 이를 재사용하는 아이디어는 매우 간단합니다.

코드로 어떻게 구현하는지 알아보죠. 여러 가지 사진 훈련된 모델이 다양한 방식으로 제공됩니다. 여기서는 구글에서 제공하는 인기있는 Inception 버전 3 모델을 사용합니다. 이 데이터셋은 ImageNet 데이터베이스에 있는 수백만 개 이상의 이미지에서 훈련되었습니다. 수십 개의 층이 있고 이미지를 천 개의 클래스로 분류할 수 있습니다. 사전 훈련된 가중치를 담은 모델을 사용할 수 있습니다.
이렇게 하려면 다음처럼 사중치를 다운로드하고 Inception V3 모델을 만들어 이 가중치를 로드합니다.

3.7 전이 학습

```
from tensorflow.keras import layers
from tensorflow.keras import Model
from tensorflow.keras.applications.inception_v3 import InceptionV3
weights_url = https://storage.googleapis.com/mledu-datasets/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5

weights_file = "inception_v3.h5"
urllib.request.urlretrieve(weights_url, weights_file)

pre_trained_model = InceptionV3(input_shape=(150, 150, 3),
                                include_top=False,
                                weights=None)

pre_trained_model.load_weights(weights_file)
```

이제 사전 훈련된 Inception 모델이 준비되었습니다. 다음 명령으로 구조를 확인해보죠

```
pre_trained_model.summary()
```

3.7 전이 학습

결과를 잠시 살펴보며 층과 이름을 확인해보세요.
mized7 층의 출력이 7 X 7 크기로 작기 때문에 이를 사용해 보겠습니다. 물론 다른 층으로 실험해봐도 좋습니다.
그 다음 다시 훈련하지 않기 위해 전체 신경망을 동결하고 mixed7의 출력을 가르키는 변수를 지정합니다. 이 지점에서 신경망을 자르기 위해서입니다. 코드는 다음과 같습니다.

```
for layer in pre_trained_model.layers:
    layer.trainable = False

last_layer = pre_trained_model.get_layer('mixed7')
print('마지막 층의 출력 크기: ', last_layer.output_shape)
last_output = last_layer.output
```

마지막 층의 출력 크기를 출력하면 결과가 7 X 7임을 알 수 있습니다. 이 결과는 이미지를 주입하면 mixed7 층에서 7 X 7 크기의 특성 맵을 출력한다는 것을 의미합니다. 다시 말하지만 꼭 mixed7 층을 선택할 필요는 없습니다. 다른 층을 실험해도 좋습니다.

3.7 전이 학습

이제 이 층 아래에 밀칩 층을 추가해보죠

```
# 출력을 펼쳐서 1차원으로 만듭니다.
x = layers.Flatten()(last_output)
# 1,204개 은닉 유닛과 렐루 활성화 함수를 사용한 완전 연결 층을 추가합니다.
x = layers.Dense(1024, activation='relu')(x)
# 분류를 위해 시그모이드 함수를 사용하는 최종 층을 추가합니다.
x = layers.Dense(1, activation='sigmoid')(x)
```

마지막 층의 출력을 밀집 층에 주입하기 위해 펼칩니다. 그 다음 1,024 개의 뉴런을 가진 밀집 층과 출력을 위해 1개의 뉴런을 가진 밀집 층을 추가합니다.
이제 사전 훈련된 모델의 입력과 x를 사용해 모델을 정의할 수 있습니다. 그 다음 이전과 동일하게 모델을 컴파일합니다.

```
model = Model(pre_trained_model.input, x)

model.compile(optimizer=RMSprop(learning_rate=0.0001),
              loss='binary_crossentropy',
              metrics=['acc'])
```

3.7 전이 학습

이 모델을 말-사람 데이터셋에 40번의 에폭 동안 훈련하면 훈련 세트에서 99% 이상의 정확도와 검증 세트에서 90% 이상의 정확도를 얻을 수 있습니다.
결과는 이전보다 훨씬 좋습니다. 하지만 튜닝으로 성능을 더 높일 수 있습니다.

3.8 다중 분류

지금까지 본 모든 예제는 두 옵션 중 하나를 선택하는 이진분류 모델을 만들었습니다. 다중 분류 모델도 거의 비슷하지만 몇 가지 차이점이 있습니다. 출력층에 시그모이드 활성화 함수를 사용하는 하나의 뉴런을 사용하는 것이 아니라, 분류하려는 클래스 개수에 해당하는 n개의 뉴런을 사용합니다. 또한 여러 개의 클래스에 적합한 손실 함수로 바꾸어야 합니다 예를 들어 이번 장에서 만든 이진 분류기의 손실 함수는 이진 클로스 엔트로피입니다. 다중 분류일 경우에는 범주형 크로스 엔트로피를 사용해야 합니다.
ImageDataGenerator를 사용해 데이터를 읽으면 레이블이 자동으로 부여됩니다. 따라서 다중 클래스일 때도 이진 분류와 동일하게 하위 디렉토리별로 레이블이 할당됩니다.
가위, 바위, 보 게임을 다루어보죠. 손 모양을 인식하려면 세 가지 종류의 이미지를 다루어야 합니다. 다행히 이를 위해 간단한 데이터셋(https://oriel.ly/VHhms)이 준비되어 있습니다.
데이터셋은 다음 두 가지로 구성됩니다. 다양한 크기, 모양, 피부색, 손톱 모양을 가진 손 이미지로 구성된 훈련세트와 이와 마찬가지로 다양하지만 훈련 세트에 없는 손 이미지로 구성된 테스트셋이 있습니다.

3.8 다중 분류

데이터셋을 사용하는 방법은 간단합니다. 다운로드하고 압축을 풀면 하위 디렉토리가 자동으로 생기므로 바로 ImageDataGenerator를 사용할 수 있습니다.

```
!wget --no-check-certificate ₩
    https://storage.googleapis.com/learning-datasets/rps.zip ₩
    -O ./rps.zip
!wget --no-check-certificate ₩
    https://storage.googleapis.com/learning-datasets/rps-test-set.zip ₩
    -O ./rps-test-set.zip
local_zip = './rps.zip'
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall('./')
zip_ref.close()
TRAINING_DIR = "./rps/"
training_datagen = ImageDataGenerator(
    rescale = 1./255,    rotation_range=40,    width_shift_range=0.2,    height_shift_range=0.2,
    shear_range=0.2,    zoom_range=0.2,    horizontal_flip=True,    fill_mode='nearest')
```

3.8 다중 분류

이 데이터셋에서 제너레이터를 만들 때 두 개 이상의 하위 디렉토리에서 ImageDataGenerator를 사용해야 하므로 class_mode 매개변수를 'categorical'로 지정해야 합니다.

```
train_generator = training_datagen.flow_from_directory(
    TRAINING_DIR,
    target_size=(150,150),
    class_mode='categorical'
)

validation_generator = validation_datagen.flow_from_directory(
    VALIDATION_DIR,
    target_size=(150,150),
    class_mode='categorical')
```

3.8 다중 분류

모델을 정의할 때 입력 층과 출력 층을 주의 깊게 살펴보아야 합니다. 입력은 데이터의 크기(여기서는 150 X 150)와 일치해야 하고 출력은 클래스 개수(여기서는 3)와 일치해야 합니다.

```
model = tf.keras.models.Sequential([
    # 입력 크기는 원하는 이미지(150x150, 3채널)와 맞아야 합니다.     # 첫 번째 합성곱 층
    tf.keras.layers.Conv2D(64, (3,3), activation='relu', input_shape=(150, 150, 3)),
    tf.keras.layers.MaxPooling2D((2, 2)),     # 두 번째 합성곱 층
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),     # 세 번째 합성곱 층
    tf.keras.layers.Conv2D(128, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),     # 네 번째 합성곱 층
    tf.keras.layers.Conv2D(128, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    # 밀집 층에 전달하가 위해 펼칩니다.
    tf.keras.layers.Flatten(),
    # 512개 뉴런을 가진 은닉층
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(3, activation='softmax')])
```

3.8 다중 분류

마지막으로 모델을 컴파일할 때 범주형 크로스 엔트로피 손실 함수를 사용해야 합니다. 이진 크로스 에트로피는 두 개 이상의 클래스에서 작동하지 않습니다.

```
model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
              metrics=['accuracy'])
```

훈련은 이전과 동일합니다.

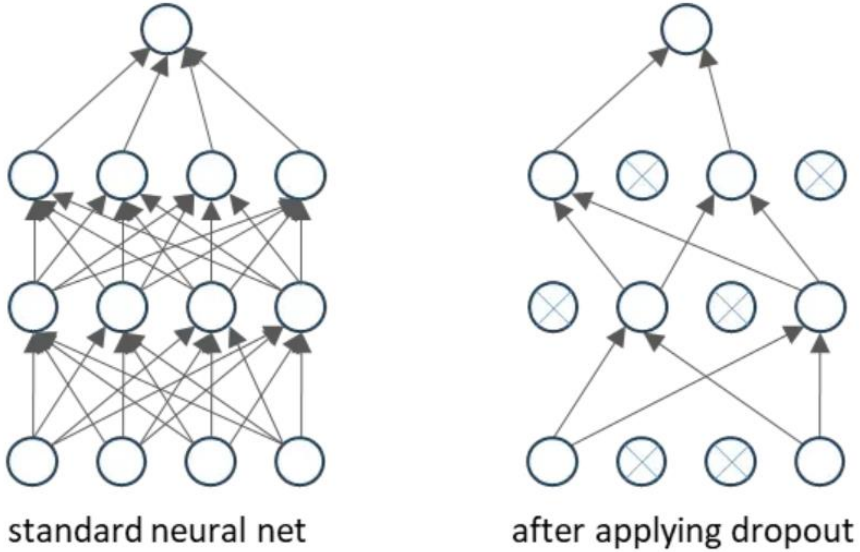
```
model.fit(train_generator,
          epochs=25,
          validation_data=validation_generator)
```

예측을 위한 코드로 조금 수정해야 합니다. 세 개의 출력 뉴런이 있기 때문에 예측된 클래스는 1에 가까운 값을 출력하고 다른 클래스는 0에 가까운 값을 출력할 것입니다. softmax 활성화 함수를 사용하므로 세 개의 예측값을 모두 더하면 1이 됩니다. 예를 들어 모델이 어떤 이미지를 보고 가위, 바위, 보 중 어떤 이미지인지 확신할 수 없다면 0.4, 0.4, 0.2와 같을 것입니다. 하지만 이미지에 대한 확신이 높다면 0.98, 0.01, 0.01을 출력할 것입니다.

ImageDataGenerator를 사용하면 디렉토리 알파벳 순서대로 클래스 레이블이 부여됩니다. 따라서 마지막 층의 출력은 순서대로 보(Paper), 바위(Rock), 가위(Scissors)가 됩니다.

3.9 드롭아웃 규제

과대적합은 신경망이 특정한 종류의 입력 데이터에 과도하게 맞춰져 있어 다른 데이터에서는 성능이 나쁜 경우였습니다. 이를 극복하는 한 가지 기법은 드롭아웃 규제(dropout regularization)입니다.
신경망을 훈련할 때 개별 뉴런은 같은 층의 다른 뉴런에 영향을 받습니다. 특히 신경망의 규모가 크다면 시간이 지남에 따라 일부 뉴런이 과도하게 전문화될 수 있습니다. 이후 층에도 이런 영향이 전달되어 전체적으로 신경망이 전문화되어 과대적합을 초래합니다. 또한 이웃한 뉴런이 비슷한 가중치와 절편을 가질 수 있습니다. 이를 모니터링하지 않으면 전체 모델이 이런 뉴런에서 감지한 특성에 특히 민감해질 수 있습니다.



훈련하는 동안 랜덤하게 일부 뉴런을 무시하면 다음 층에 있는 뉴런에 미치는 영향을 일시적으로 막을 수 있습니다.

3.9 드롭아웃 규제

드롭아웃을 사용하면 뉴런이 과도하게 특화될 가능성을 줄입니다. 신경망은 여전히 같은 수의 파라미터를 학습하지만 일반화 성능은 나아집니다. 즉 다른 종류의 입력도 안정적으로 처리할 것입니다.
텐서플로에서는 드롭아웃을 사용하려면 다음처럼 케라스 층을 추가합니다.

```
tf.keras.layers.Dropout(0.2)
```

이전 층에 있는 뉴런 중에서 지정된 양(여기서는 20%)만큼 랜덤하게 드롭아웃합니다. 현재 신경망에 적절한 퍼센트를 찾으려면 실험이 필요할 수 있습니다.
이를 패션 MNIST 분류기에 적용해보죠 다음과 같이 신경망에 층을 더 추가합니다.

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(256, activation=tf.nn.relu),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(64, activation=tf.nn.relu)
,    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

3.9 드롭아웃 규제

이 신경망을 동일한 데이터에서 같은 에폭 횟수 동안 훈련하면 훈련 세트 정확도가 89%로 줄어듭니다. 검증 세트의 정확도는 거의 비슷하게 88%를 달성합니다. 두 값이 이전보다 더 가깝습니다. 드롭아웃을 추가하면 과대적합이 있다는 것을 나타낼 뿐만 아니라 신경망이 훈련 데이터에 과도하게 특화되지 않도록 만드는 데에도 도움이 됩니다.
신경망을 만들 때 훈련 셋트에서 좋은 결과가 항상 옳은 것이 아니라는 것을 기억하세요. 이는 과대적합의 신호일 수 있습니다.

Reference

개발자를 위한 머신러닝&딥러닝, Laurence Moroney, 박해선, 한빛미디어
https://github.com/rickiepark/aiml4coders
https://github.com/lmoroney/tfbook
https://sputnik-kr.tistory.com/237
https://compmath.korea.ac.kr/deeplearning/ConvolutionNN.html
https://hyun-tori.tistory.com/79
https://thinking-potato.tistory.com/40

외 본문 링크