이번 절에서는 텐서플로, 케라스 **API**를 사용해 **7**장에서 만들었던 패션 **MNIST** 데이터를
합성곱 신경망으로 분류할 것이다.

패션 MNIST 데이터 불러오기

`keras.datasets.fashion_mnist.load_data()`

패션 **MNIST** 데이터를 불러오고 적절히 전처리하기

- 데이터 스케일을 **0-255 -> 0-1** 로 바꾸고
- 훈련 세트와 검증 세트로 나누기

한 가지 작업이 다르다.
완전 연결 신경망에서는 입력 이미지를 밀집층에 연결하기 위해 일렬로 펼쳐야 한다.
이 작업을 위해 넘파이의 **reshape** 함수나 **Flatten** 클래스를 사용했었다.
합성곱 신경망은 **2**차원 이미지를 그대로 사용하기 때문에 **1**차원으로 변환할 필요가 없다.

다만 **8**장 **1**절에서 이야기했듯이 입력 이미지는 항상 깊이**(** 채널**)** 차원이 있어야 한다.
흑백 이미지의 경우 채널 차원이 없는 **2**차원 배열이지만 **Conv2D** 층을 사용하기 때문에
마지막에 이 채널 차원을 추가해야 한다.
넘파이 **reshape()** 메서드를 사용해 전체 배열 차원을 그대로 유지하면서 마지막에 차원을 간단히 추가할 수 있다.

In [1]:

```
from tensorflow import keras
from sklearn.model_selection import train_test_split

(train_input, train_target), (test_input, test_target) = keras.datasets.fashion_mnist.lo
ad_data()

train_scaled = train_input.reshape(-1, 28, 28, 1) / 255.0

train_scaled, val_scaled, train_target, val_target = train_test_split(
    train_scaled, train_target, random_state = 42, test_size = 0.2
)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-l
abels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-i
mages-idx3-ubyte.gz
26421880/26421880 [==============================] - 2s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-la
bels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-im
ages-idx3-ubyte.gz
4422102/4422102 [==============================] - 0s 0us/step
```

이제 **(48000, 28, 28)** 크기의 **train_input**이 **(48000, 28, 28, 1)** 크기의 **4**차원 배열이 되었다.

- **48000 :** 입력 데이터 개수
- **28** ***28*** **1 :** 입력 피처의 형식 **(** 그런데 왜 **28 * 28**로 안하고,, **??** **)**

그 외 다른 작업은 **7**장에서 했던 것과 같다.
데이터를 준비했으니, 합성곱 신경망을 만들어 보자.

합성곱 신경망 만들기

전형적인 합성곱 신경망의 구조는 합성곱 츠응로 이미지에서 특징을 감지한 후
밀집층으로 클래스에 따른 분류 확률을 계산한다.
케라스의 **Sequential** 클래스를 사용해 순서대로 이 구조를 정의해 보겠다.

먼저 **Sequential** 클래스의 객체를 만들고 첫 번째 합성곱 층인 **Conv2D**를 추가한다.
이 클래스는 다른 층 클래스와 마찬가지로 **keras.layers** 패키지 아래에 있다.
여기에서는 이전 장에서 보았던 모델의 **add()** 메서드를 사용해 층을 하나씩 차례대로 추가하겠다.

In [2]:

```
model = keras.Sequential()
model.add(keras.layers.Conv2D(32, kernel_size = 3, activation = 'relu', padding = 'same'
, input_shape = (28, 28, 1)))
```

이 코드의 매개변수를 자세히 살펴보겠다.
이 합성곱 층은 **32**개의 필터를 사용한다.
커널의 크기는 **(3, 3)**이고, **ReLU** 활성화 함수의 **same padding**을 사용한다.

완전 연결 신경망에서처럼 케라스 신경망 모델의 첫 번째 층에서 입력의 차원을 지정해 주어야 한다.
앞서 패션 **MNIST** 이미지를 **(28, 28)**에서 **(28, 28, 1)**로 변경했었다.
input_shape 매개변수를 이 값으로 지정한다.

그 다음 풀링 층을 추가한다. 케라스는 최대 풀링과 평균 풀링을 **keras.layers** 패키지 아래에
MaxPooling2D, AveragePooling2D 클래스로 제공한다.
전형적인 풀링 크기인 **(2, 2)** 풀링을 사용해 보자.
Conv2D 클래스의 **kernel_size**처럼 가로세로 크기가 같으면 정수 하나로 지정할 수 있다.

In [3]:

```
model.add(keras.layers.MaxPooling2D(2, 2))
```

패션 **MNIST** 이미지가 **(28, 28)** 크기에 세임 패딩을 적용했기 때문에
합성곱 층에서 출력된 특성 맵의 가로세로 크기는 입력과 동일하다.

그 다음 **(2, 2)** 풀링을 사용했기 때문에 특성 맵의 크기는 절반으로 줄어든다.
합성곱 층에서 **32**개의 필터를 사용했기 때문에 이 특성 맵의 크기는 **32**가 된다.
따라서 최대 풀링을 통화한 특성 맵의 크기는 **(14, 14, 32)**가 될 것이다.

나중에 각 층의 출력 크기를 **summary()** 메서드로 확인해 보겠다.

첫번째 합성곱-풀링 층 다음에 두 번째 합성곱 풀링 층을 추가해 보겠다.
두 번째 합성곱-풀링 층은 첫 번째와 거의 동일하다. 필터의 개수를 **64**개로 늘린 점만 다르다.

In [5]:

```
model.add(keras.layers.Conv2D(64, kernel_size = 3, activation = 'relu', padding = 'same'
))
model.add(keras.layers.MaxPooling2D(2))
```

첫 번째 합성곱-풀링 층과 마찬가지로 이 합성곱 층은 세임 패딩을 사용한다.
따라서 입력의 가로-세로 크기를 줄이지 않는다.
이어지는 풀링 층에서 이 크기를 절반으로 줄인다.
64개의 필터를 사용했으므로 최종적으로 만들어지는 특성 맵의 크기는 **(7, 7, 64)**가 될 것이다.

이제 이 **3**차원 특성 맵을 일렬로 펼칠 차례이다.
이렇게 하는 이유는 마지막에 **10**개의 뉴런을 가진 **(**밀집**)**출력응에서 확률을 계산하기 때문

여기에서는 특성 맵을 일렬로 펼쳐서 바로 출력층에 전달하지 않고
중간에 하나의 밀집 은닉층을 하나 더 두도록 하겠다.

즉 **Flatten** 클래스 다음에 **Dense** 은닉층, 마지막 **Dense** 출력층의 순서대로 구성한다.

In [7]:

```
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(100, activation = 'relu'))
model.add(keras.layers.Dropout(0.4))
model.add(keras.layers.Dense(10, activation = 'softmax'))
```

은닉 층과 출력 층 사이에 **Dropout**을 넣었다. 드롭아웃 층이 은닉층의 과대적합을 막아
성능을 더 개선해 줄 것이다.
은닉층은 **100**개의 뉴런을 사용하고, 확성화 함수는 합성곱 층과 마찬가지로 **relu** 함수를 사용

패션 **MNIST** 데이터셋은 클래스 **10**개를 분류하는 다중 분류 문제이므로
마지막 층의 활성화 함수는 소프트맥스를 활용한다.

이렇게 합성곱 신경망의 구성을 마쳤다.
앞 절에서 커널, 패딩, 풀링 등을 잘 이해했다면, 케라스 **API**를 사용해
손쉽게 다양한 구성을 실험해 볼 수 있다.

케라스 모델의 구성을 마쳤으니 **summary()** 메서드로 모델 구조를 출력해 보겠다.

In [8]:

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 32)        320

 max_pooling2d (MaxPooling2D  (None, 14, 14, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 14, 14, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 7, 7, 64)         0
 2D)

 flatten (Flatten)           (None, 3136)              0

 dense (Dense)               (None, 100)               313700

 flatten_1 (Flatten)         (None, 100)               0

 dense_1 (Dense)             (None, 100)               10100

 dropout (Dropout)           (None, 100)               0

 dense_2 (Dense)             (None, 10)                1010

=================================================================
Total params: 343,626
Trainable params: 343,626
Non-trainable params: 0
_________________________________________________________________
```

summary() 메서드의 출력 결과를 보면 합성곱 층과 풀링 효과가 잘 나타나 있다.
첫 번째 합성곱 층을 통과하면서 특성 맵의 깊이는 **32**가 되고, 두 번째 합성곱에서 특성 맵의 크기가 **64**로 늘어난다.

반면 특성 맵의 가로세로 크기는 첫 번째 풀링 층에서 절반으로 줄어들고, 두 번째 풀링 층에서 다시 절반으로 줄어든다.

따라서 최종 맵의 크기는 **(7, 7, 64)**이다.

완전 연결 신경망에서 했던 것처럼 모델 파라미터의 개수를 계산해 보자.
첫 번째 합성곱 층은 **32**개의 필터를 가지고 있고 크기가 **(3, 3)**, 깊이가 **1**이다.
또 필터마다 하나의 절편이 있다.

따라서 총 **3*31*32 + 32 = 320**개의 파라미터가 있다.

두 번째 합성곱 층은 **64**개의 필터를 사용하고, 크기가 **(3, 3)**, 깊이가 **32**개이다.
역시 필터마다 하나의 절편이 있다.
총 **3 *3* 32 * 64 + 64 = 18,496**개의 파라미터가 있다.
층의 구조를 잘 이해하고 있는지 확인하려면 이렇게 모델 파라미터 개수를 계산해 보아야 한다.

Flatten 클래스에서 **(7, 7, 64)**크기의 특성 맵을 **1**차원 배열로 펼치면 **(3136,)** 크기의 배열이 된다.
이를 **100**개의 뉴런과 완전히 연결해야 하므로 은닉층의 모델 파라미터 개수는 **3, 136*100 + 100 = 313,700**개이다.

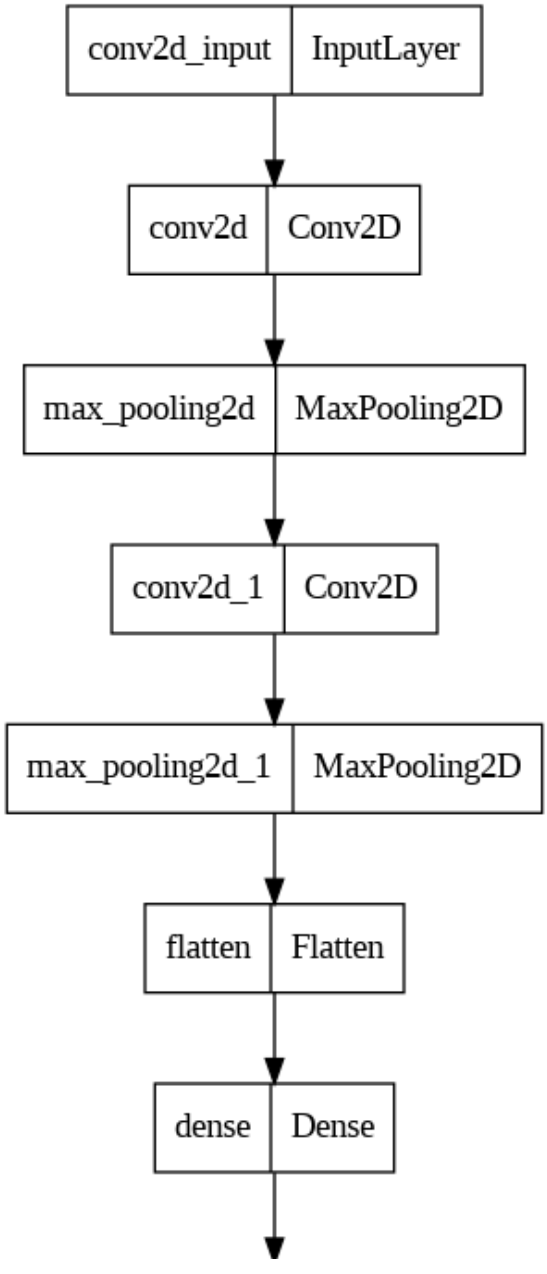
마찬가지 방식으로 계산하면 마지막 출력층의 모델 파라미터 개수는 **1,010**개이다.

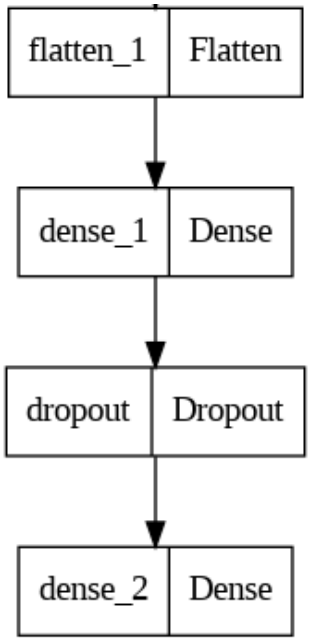
케라스는 **summary()** 메서드 외에 층의 구성을 그림으로 표현해 주는 **plot_model()** 함수를
keras.utils 패키지에서 제공한다.
이 함수 앞에서 만든 **model** 객체를 넣어 호출해 보자.

In [10]:

```
keras.utils.plot_model(model)
```

Out[10]:



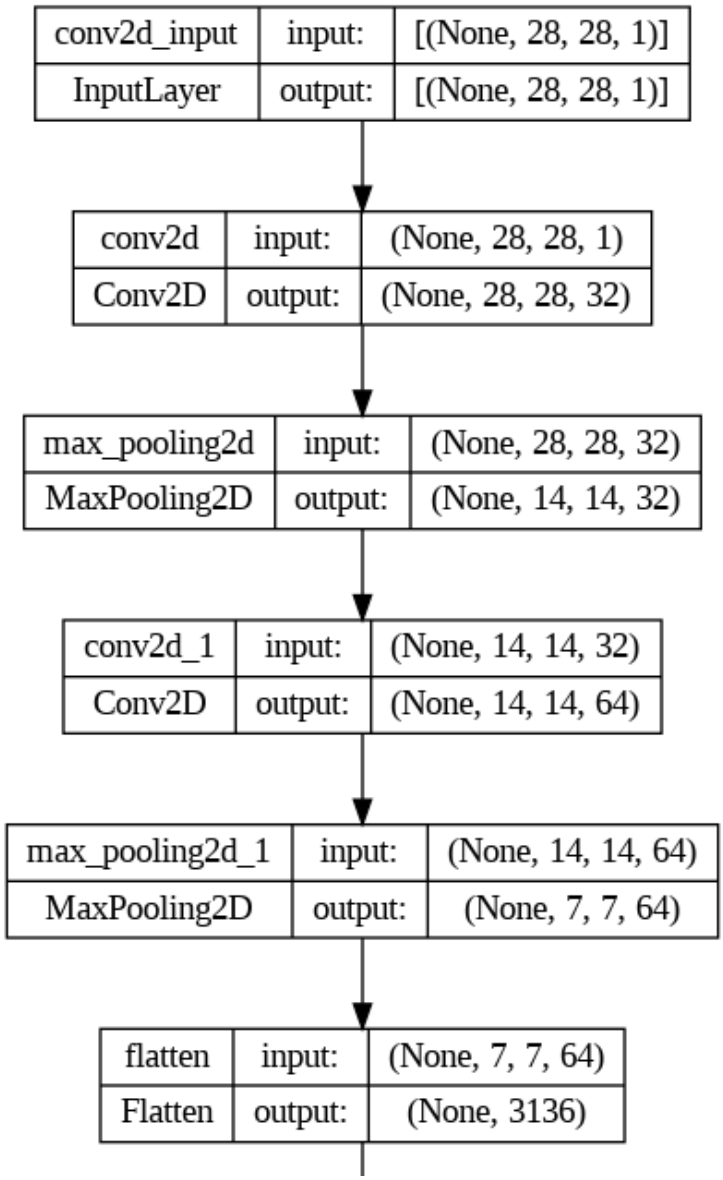


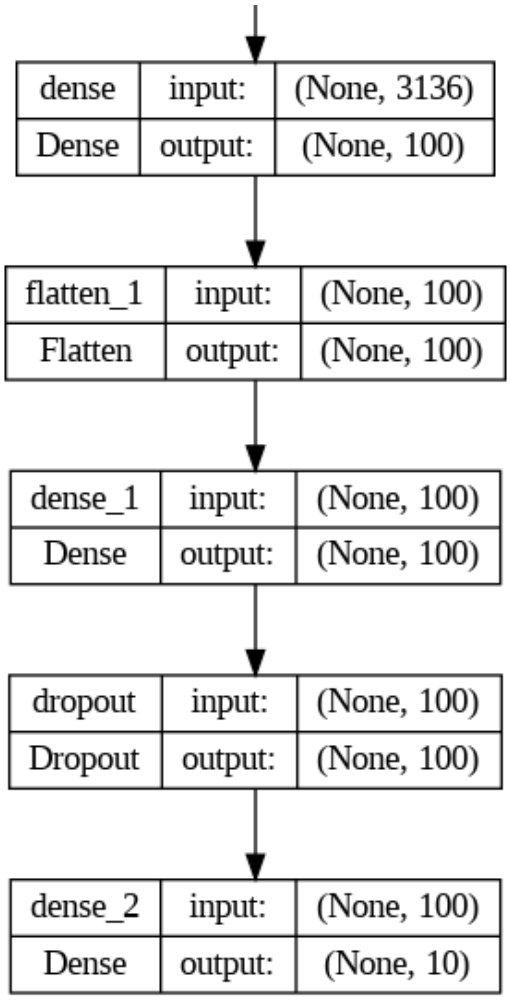
plot_model() 함수의 **show_shapes** 매개변수를 **True**로 설정하면 이 그림에 입력과 출력의 크기를 표시해 준다.
또 **to_file** 매개변수에 파일 이름을 지정하면 출력한 이미지를 파일로 저장한다.
dpi 매개변수로 해상도를 지정할 수도 있다.
이 옵션들을 사용해 다시 그래프를 그려 보자.

In [11]:

```
keras.utils.plot_model(model, show_shapes = True)
```

Out[11]:





이제 모델을 컴파이하고 훈련해 보자.

모델 컴파일과 훈련

케라스 **API**의 장점은 딥러닝 모델의 종류나 구성 방식에 상관없이 컴파일과 훈려 과정이 같다는 것이다.
다음 코드는 **7**장 **3**절에서 사용했던 완전 연결 신경망 모델을 컴파일하고 훈련하는 코드와 거의 같다.

Adam 옵티마이저를 사용하고, **ModelCheckpoint** 콜백과 **EarlyStopping** 콜백을 함께 사용해
조기 종료 기법을 구현한다.

In [12]:

```
model.compile(optimizer = 'adam', loss='sparse_categorical_crossentropy', metrics='accur
acy')
checkpoint_cb = keras.callbacks.ModelCheckpoint('best-cnn-model.h5', save_best_only = Tru
e)
early_stopping_cb = keras.callbacks.EarlyStopping(patience = 2, restore_best_weights = T
rue)
history = model.fit(train_scaled, train_target, epochs = 20,
                    validation_data = (val_scaled, val_target),
                    callbacks = [checkpoint_cb, early_stopping_cb])
```

```
Epoch 1/20
1500/1500 [==============================] - 98s 65ms/step - loss: 0.4952 - accuracy: 0.8
229 - val_loss: 0.3498 - val_accuracy: 0.8737
Epoch 2/20
1500/1500 [==============================] - 66s 44ms/step - loss: 0.3128 - accuracy: 0.8
894 - val_loss: 0.2772 - val_accuracy: 0.9002
Epoch 3/20
1500/1500 [==============================] - 65s 43ms/step - loss: 0.2627 - accuracy: 0.9
049 - val_loss: 0.2664 - val_accuracy: 0.8994
Epoch 4/20
1500/1500 [==============================] - 65s 44ms/step - loss: 0.2282 - accuracy: 0.9
178 - val_loss: 0.2450 - val_accuracy: 0.9119
Epoch 5/20
```

```
Epoch 5/20
1500/1500 [==============================] - 65s 43ms/step - loss: 0.1997 - accuracy: 0.9
255 - val_loss: 0.2264 - val_accuracy: 0.9203
Epoch 6/20
1500/1500 [==============================] - 64s 43ms/step - loss: 0.1777 - accuracy: 0.9
348 - val_loss: 0.2382 - val_accuracy: 0.9170
Epoch 7/20
1500/1500 [==============================] - 64s 43ms/step - loss: 0.1597 - accuracy: 0.9
413 - val_loss: 0.2359 - val_accuracy: 0.9175
```

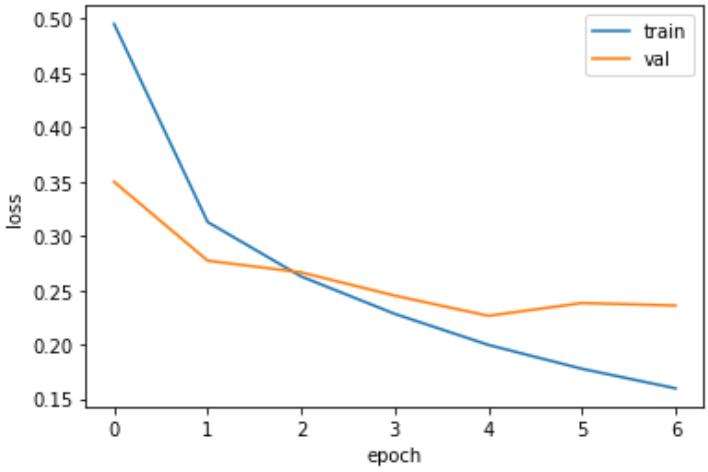
얼핏 보아도 훈련 세트의 정확도가 이전보다 훨씬 좋아진 것을 확인할 수 있다. **(80%**대에서 **90%**대로 **)**
손실 그래프를 그래서 조기 종료가 잘 이루어졌는지 확인해 보자.

In [13]:

```
import matplotlib.pyplot as plt

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend(['train', 'val'])

plt.show()
```



검증 세트에 대한 손실이 점차 감소하다가 정체되기 시작하고 훈련 세트에 대한 손실은 점점 더 낮아지고 있다.

EarlyStopping 클래스에서 **restore_best_weights** 매개변수를 **True**로 지정했으므로
현재 **model** 객체가 최적의 모델 파라미터로 복원되어 있다.
즉 **Model Checkpoint** 콜백이 저장한 **best-cnn-model.h5** 파일을 다시 읽을 필요가 있다.

이번에는 세트에 대한 성능을 평가해 보자.

In [14]:

```
model.evaluate(val_scaled, val_target)
```

```
375/375 [==============================] - 5s 14ms/step - loss: 0.2264 - accuracy: 0.9203
```

Out[14]:

```
[0.22644449770450592, 0.9203333258628845]
```

이 결과는 **fit()** 메서드의 출력 중 아홉 번째 에포크의 출력과 동일하다.
EarlyStopping 콜백이 **model** 객체를 최상의 모델 파라미터로 잘 복원한 것 같다.

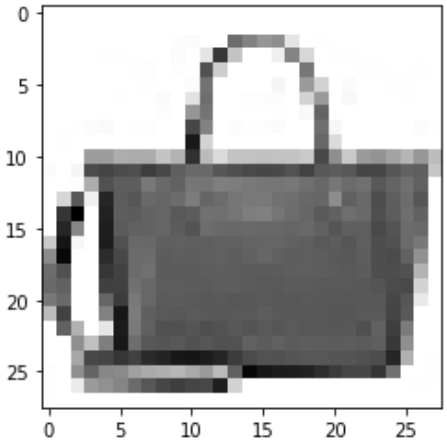
7장에서 잠깐 소개했던 **predict()** 메서드를 사용해 훈련된 모델을 사용하여 새로운 데이터에 대해 예측을 만들어 보겠다.
여기에서는 편의상 검증 세트의 첫 번째 샘플을 처음 본 이미지라고 가정한다.

맷플롯립에서는 흑백 이미지에 깊이 차원은 없다. 따라서 **(28, 28, 1)** 크기를
(28. 28)로 바꾸어 출력해야 한다.

첫 번째 샘플 이미지를 먼저 확인해 보자.

In [15]:

```
plt.imshow(val_scaled[0].reshape(28, 28), cmap = 'gray_r')
plt.show()
```



핸드백 이미지 같다. 모델은 이 이미지에 대한 어떤 예측을 만드는 지 확인해 보자.
predict() 메서드는 **10**개의 클래스에 대한 예측 확률을 출력한다.

In [16]:

```
preds = model.predict(val_scaled[0:1])
print(preds)
```

```
1/1 [==============================] - 0s 165ms/step
[[4.6119236e-14 4.4928190e-21 1.5375218e-15 6.6822546e-16 1.9195668e-15
  1.0291835e-13 5.7312736e-14 6.3830485e-14 1.0000000e+00 2.0297486e-16]]
```

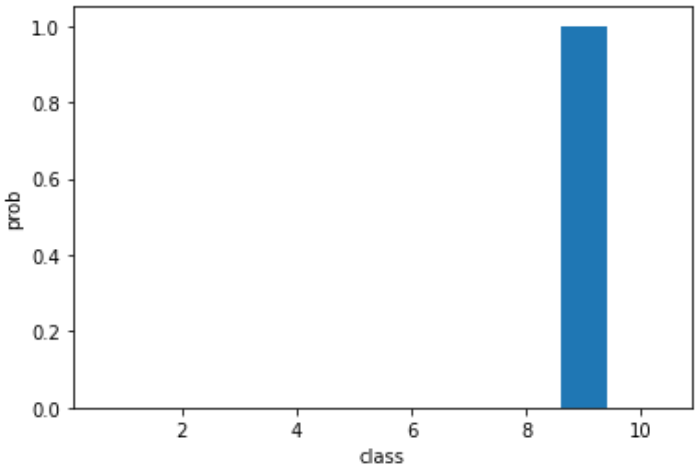
[0:1]과 같이 슬라이싱을 사용했다.
케라스의 **fit(), predict(), evalate()** 메서드는 모두 입력의 첫 번째 차원이 배치 차원일 것으로 기대한다.
따라서 샘플 하나를 전달할 때 **(28, 28, 1)**이 아니라 **(1, 28, 28, 1)** 크기로 전달해야 한다.
배열 슬라이싱은 인덱싱과 다르게 선택된 원소가 하나이더라도 전체 차원이 유지되어 **(1, 28, 28, 1)**크기를 만든다.

출력 결과를 보면 아홉 번째 값이 **1**이고 다른 값은 거의 **0**에 가깝다.
이를 막대그래프로 보면 확실히 느낄 수 있다.

In [17]:

```
plt.bar(range(1, 11), preds[0])
plt.xlabel('class')
plt.ylabel('prob')

plt.show()
```



다른 클래스의 값들은 사실상 **0**이다. 아홉 번째 클래스가 실제로 무엇이였는지는 패션 **MNIST** 데이터셋의 정의를 참고해

야 한다.
7장에서 패션 **MNIST** 데이터셋의 레이블을 보았었다.
여기에서는 파이썬에서 레이블을 다루기 위해 리스트로 저장하겠다.

In [18]:

```
classes = ['티셔츠', '바지', '스웨터', '드레스', '코트', '샌달', '셔츠', '스니커즈', '가방', '앵
클 부츠']
```

클래스 리스트가 있다면 레이블을 출력하기 쉽다. **preds** 배열에서 가장 큰 인덱스를 찾아
classes 리스트의 인덱스로 사용하면 된다.

In [19]:

```
import numpy as np
print(classes[np.argmax(preds)])
```

가방

위의 그림 예시 샘플을 가방으로 잘 예측한 것 같다.
합성곱 신경망을 만들고, 훈련하여 새로운 샘플에 대해 예측을 수행하는 방법도 알아보았다.
마지막으로 맨 처음에 떼어 놓았던 테스트 세트로 합성곱 신경망의 일반화 성능을 가늠해 보겠다.
즉, 이 모델을 실전에 투입하였을 때 얻을 수 있는 예상 성능을 측정해 보자.

훈련 세트와 검증 세트에서 했던 것처럼 픽셀 값의 범위를 **0~1** 사이로 바꾸고
이미지 크기를 **(28, 28)**에서 **(28, 28, 1)** 크기로 바꾸겠다.

In [22]:

```
test_scaled = test_input.reshape(-1, 28, 28, 1) / 255.0
```

그 다음 **evaluate()** 메서드로 테스트 세트에 대한 성능을 측정한다.

In [23]:

```
model.evaluate(test_scaled, test_target)
```

```
313/313 [==============================] - 4s 13ms/step - loss: 0.2506 - accuracy: 0.9142
```

Out[23]:

```
[0.2505660057067871, 0.9142000079154968]
```

테스트 세트에서의 점수는 검증 세트보다 조금 더 작다.
이 모델을 실전에 투입하여 패션 아이템을 분류한다면 약 **91%**의 성능을 기대할 수 있을 것이다.