손실 곡선

2절에서 fit() 메서드로 모델을 훈련하면 마지막에 다음과 같은 메세지가 있다.

<keras.callbacks.History at 0x7fcf4ab62df0>

이 메세지는 fit() 메서드의 실행 결과를 출력한 것이다.
케라스의 fit() 메서드는 `History()` 클래스 객체를 반환한다.

History 객체에는 훈련 과정에서 계산한 지표, 즉 손실과 정확도 값이 저장되어 있다.
이 값을 사용하면 그래프를 그릴 수 있을 것이다.

먼저 이전 절에서 사용했던 것과 같이 패션 MNIST 데이터셋을 적재하고 훈련 세트와 검증 세트로 나누자.

In [1]:
```python
from tensorflow import keras

(train_input, train_target), (test_input, test_target) = keras.datasets.fashion_mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 [==============================] - 0s 0us/step
```

In [2]:
```python
from sklearn.model_selection import train_test_split

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

그 다음 모델을 만들겠다. 그런데 이전 절과는 달리 모델을 만드는 간단한 함수를 정의하겠다.
이 함수는 하나의 매개변수를 가진다. 먼저 코드를 보자.

In [9]:
```python
def model_fn(a_layer=None):
    model = keras.Sequential()
    model.add(keras.layers.Flatten(input_shape=(28, 28)))
    model.add(keras.layers.Dense(100, activation='relu'))
    if a_layer:
        model.add(a_layer)
    model.add(keras.layers.Dense(10, activation='softmax'))
    return model
```

은닉층을 추가하는 함수이다. 케라스 층을 추가하면, 은닉층 뒤에 파라미터로 제공하는 또다른 은닉층을 추가하는 함수이다.

여기서는 a_layer 매개변수로 층을 추가하지 않고, 단순하게 model_fn() 함수를 호출한다.

In [15]:
```python
model = model_fn()

model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_6 (Flatten)         (None, 784)               0

 dense_8 (Dense)             (None, 100)               78500

 dense_9 (Dense)             (None, 10)                1010

=================================================================
Total params: 79,510
Trainable params: 79,510
Non-trainable params: 0
_________________________________________________________________
```

이전 절과 동일하게 모델을 훈련하지만 fit() 메서드의 결과를 history() 변수에 담아 보겠다.

In [16]:
```python
model.compile(loss = 'sparse_categorical_crossentropy', metrics = 'accuracy')
history = model.fit(train_scaled, train_target, epochs=5, verbose=0)
```

verbose 매개변수는 훈련 과정 출력을 조정한다. 기본값 : 1

- 1 : 이전 절에서처럼 진행 막대와 함께 손실 등의 지표가 출력됨
- 2 : 진행 막대를 빼고 진행
- 0 : 진행 과정을 나타내지 않음

history 객체에는 훈련 측정값이 들어 있는 history 딕셔너리가 들어 있다. 이 딕셔너리에
어떤 값이 들어 있는지 확인해 보자.

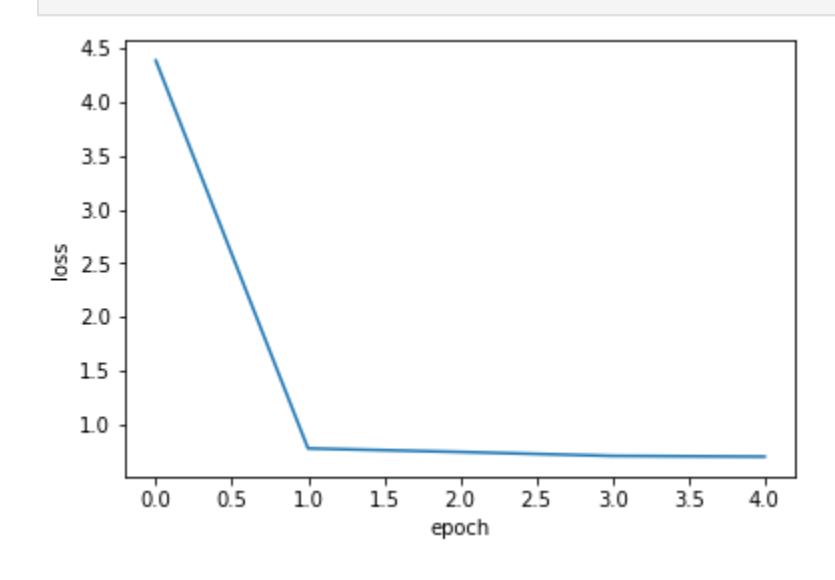
In [17]:
```python
print(history.history.keys())
```

```
dict_keys(['loss', 'accuracy'])
```

손실과 정확도가 들어 있다. 이전 절에서 언급했듯이 케라스는 기본적으로 에포크마다 손실을 계산한다.
정확도는 compile() 메서드에서 metrics 매개변수에 accuracy를 추가했기 때문

history 속성에 포함된 손실과 정확도는 에포크마다 계산한 값이 순서대로 나열된 단순한 리스트
matplotlib을 통해 쉽게 그래프로 그릴 수 있다.

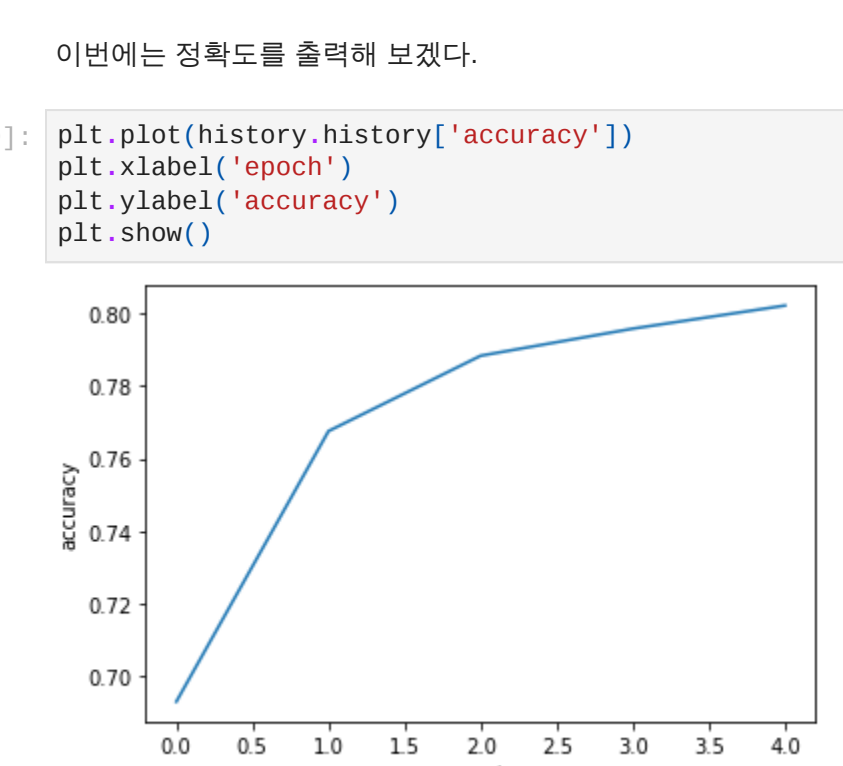
In [18]:
```python
import matplotlib.pyplot as plt

plt.plot(history.history['loss'])
plt.xlabel('epoch')
plt.ylabel('loss')

plt.show()
```



파이썬 리스트의 인덱스는 0부터 시작하므로 5개의 에포크가 0에서부터 4까지 x축에 표현된다.

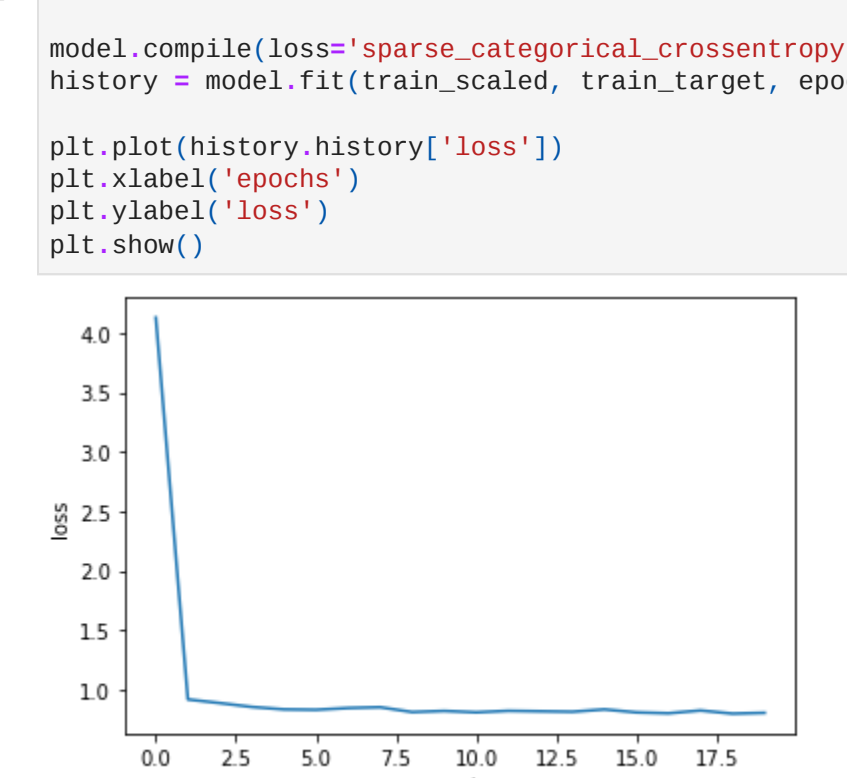
이번에는 정확도를 출력해 보겠다.

In [19]:
```python
plt.plot(history.history['accuracy'])
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```



확실히 에포크마다 손실이 감소하고 정확도가 향상된다. 그렇다면 에포크를 늘려 저 훈련하면
계속 손실이 감소할 것이라고 예상된다.
에포크를 20으로 늘려 모델을 훈련하고 손실 그래프를 그려 보자.

In [21]:
```python
model = model_fn()

model.compile(loss='sparse_categorical_crossentropy', metrics='accuracy')
history = model.fit(train_scaled, train_target, epochs=20, verbose=0)

plt.plot(history.history['loss'])
plt.xlabel('epochs')
plt.ylabel('loss')
plt.show()
```



예상대로 손실이 잘 감소한다. 이전보다 더 나은 모델을 훈련한 것일까 ?
이전에 배웠던 것 중에 놓친 것이 있지는 않을까 ?

검증 손실

4장에서 확률적 경사 하강법을 사용했을 때 과대/과소적합과 에포크 사이의 관계를 알아보았었다.
인공 신경망은 모두 일종의 경사 하강법을 사용하기 때문에 동일한 개념이 여기에도 적용된다.

에포크에 따른 과대적합과 과소적합을 파악하려면 훈련 세트에 대한 점수뿐만 아니라 검증 세트에 대한 점수도 필요하다.

에포크마다 검증 손실을 계산하기 위해 케라스 모델의 fit() 메서드에 검증 데이터를 전달할 수 있다.
다음처럼 validation_data 매개변수에 검증에 사용할 입력과 타깃값을 튜플로 만들어 전달한다.

In [25]:
```python
model = model_fn()
model.compile(loss='sparse_categorical_crossentropy', metrics='accuracy')
history = model.fit(train_scaled, train_target, epochs=20, verbose = 0, validation_data = (val_scaled, val_target))
```

위의 과정은 실행하는 데 시간이 걸린다. GPU를 사용하면 조금 더 빠르다.

반환된 history.history 딕셔너리에 어떤 값이 들어있는 지 키를 확인해 보자.

In [26]:
```python
print(history.history.keys())
```

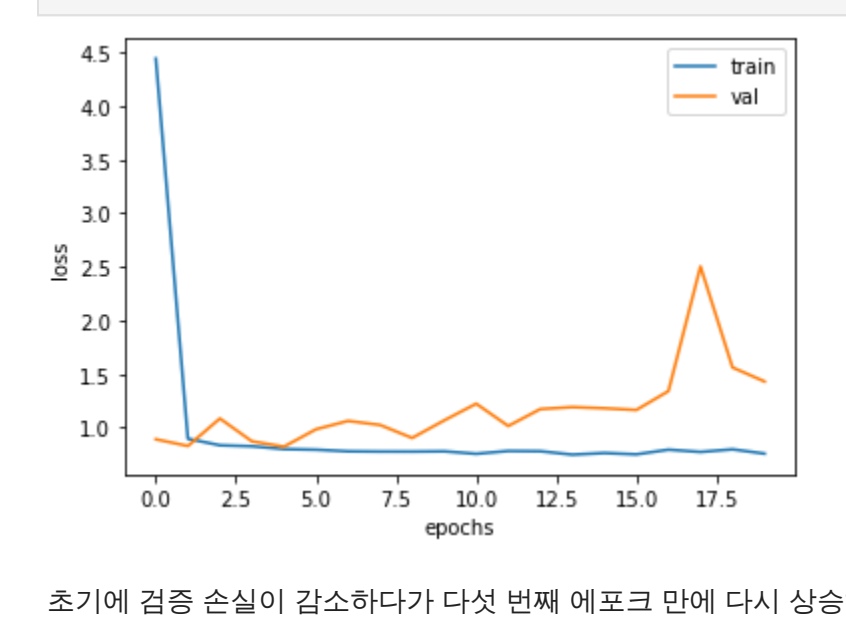
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

검증 데이터에 대한

- 손실 값 : val_loss
- 정확도 ; val_accuracy
  에 담겨 있다.

과대/과소적합 문제를 조사하기 위해 훈련 손실과 검증 손실을 한 그래프에 그려서 비교해 보자.

In [27]:
```python
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.xlabel('epochs')
plt.ylabel('loss')
plt.legend(['train', 'val'])
plt.show()
```



초기에 검증 손실이 감소하다가 다섯 번째 에포크 만에 다시 상승하기 시작했었다.
과대적합 모델이 만들어졌었던 경우가 있었다.

검증 손실이 상승하는 시점을 가능한 뒤쪽으로 늦추면 검증 세트에 대한 손실이 줄어들 뿐만 아니라 검증 세트에 대한 정확도도 늘어날 것이다.

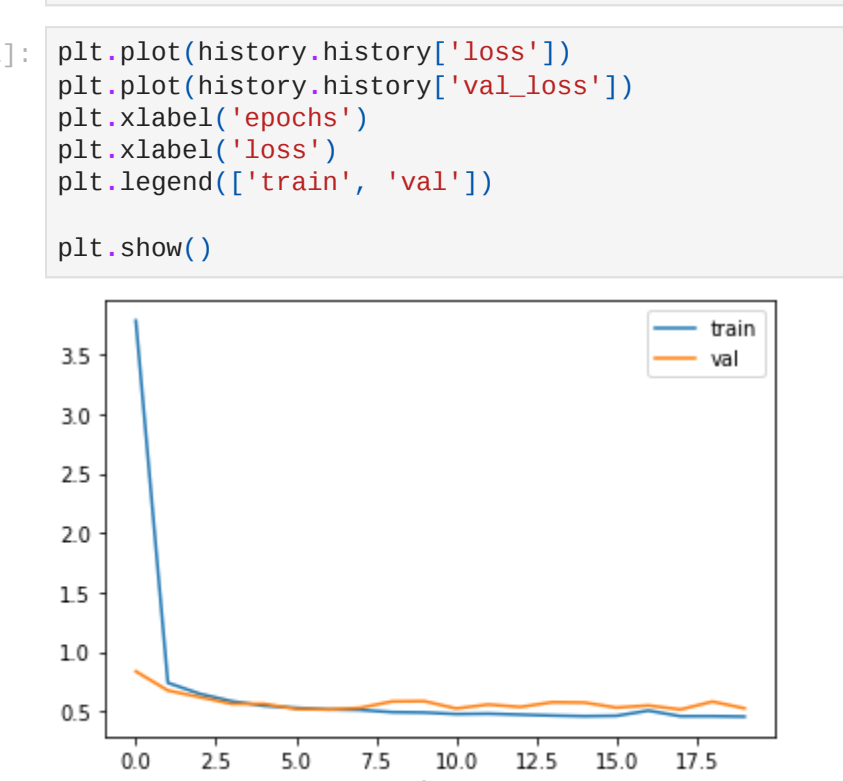
3장에서의 규제 방식 대신에 신경망에서의 특화된 규제 방법을 다음 섹션에서 다루어 보겠다.

기본 RMSprop 옵티마이저는 많은 곳에서 작동한다. 그러나 다른 옵티마이저를 테스트 해본다면
Adam이 좋은 선택이다. 적응적 학습률을 적용하기 때문에 에포크가 진행되면서 학습률의 크기를 조정할 수 있다.

Adam 옵티마이저를 적용해 보고 훈련 손실과 검증 손실을 다시 그려 보자.

In [30]:
```python
model = model_fn()
model.compile(optimizer = 'adam', loss='sparse_categorical_crossentropy', metrics='accuracy')
history = model.fit(train_scaled, train_target, epochs = 20, verbose=0, validation_data = (val_scaled, val_target))
```

In [31]:
```python
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.xlabel('epochs')
plt.xlabel('loss')
plt.legend(['train', 'val'])

plt.show()
```



Adam 옵티마이저가 이 데이터셋에 잘 맞는다.
더 나은 손실곡선을 얻으려면 학습률을 조정해서 다시 시도해 볼 수 있다.
https://wikidocs.net/36033

드롭아웃

훈련 과정에서 층에 있는 일부 뉴런을 랜덤하게 꺼서(죽여서) 과대적합을 막는 방법

각 에포크마다 랜덤으로 비율에 맞춰 노드(뉴런)을 죽인다. 따라서 각 데이터마다
꺼지는 뉴런의 종류가 제각기 다르다. 랜덤하게 드롭아웃된다.
얼마나 많은 뉴런을 드롭할 지는 우리가 정해야 할 또다른 하이퍼파라미터이다.

드롭아웃이 왜 과대적합을 막을까 ?

- 특정 뉴런에 과대 의존하는 것을 막을 수 있다.
- 모든 입력에 대해 주의를 기울여야 한다.
- 일부 뉴런의 출력이 없을 수 있다는 것을 감안하면 이 신경망은 더 안정적인 예측을 만들 수 있다.

드롭아웃을 적용해 훈련한느 것은 마치 2개의 신경망을 앙상블 하는 것처럼 상상할 수 있다.

앙상블은 과대적합을 막을 수 있는 아주 좋은 기법이다.

케라스에서는 드롭아웃을 keras.layers 패키지 아래 Dropout 클래스로 제공한다.
어떤 층의 드롭아웃을 두어 이 층의 출력을 랜덤하게 0으로 만든다.
드롭아웃이 층처럼 사용되지만 훈련되는 모델 파라미터는 없다.

그럼 앞서 정의한 model_fn() 함수에 드롭아웃 객체를 전달하여 층을 추가해 보자.
여기에서는 30% 정도를 드롭아웃한다. 만들어진 모델의 summary() 메서드를 사용해 드롭아웃 층이 잘 추가되었는지 확인해 보자.

In [32]:
```python
model = model_fn(keras.layers.Dropout(0.3))
model.summary()
```

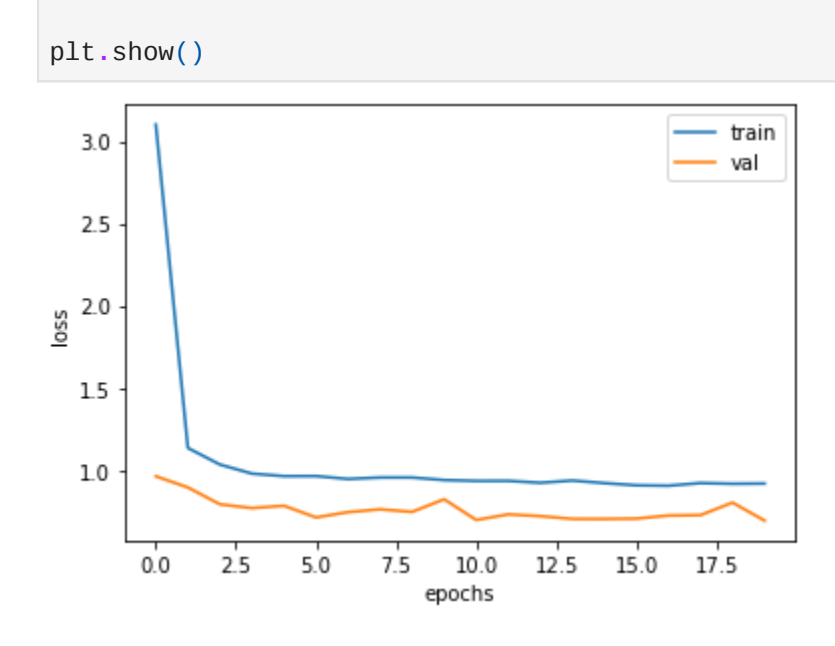
```
Model: "sequential_12"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_12 (Flatten)        (None, 784)               0

 dense_20 (Dense)            (None, 100)               78500

 dropout (Dropout)           (None, 100)               0

 dense_21 (Dense)            (None, 10)                1010

=================================================================
Total params: 79,510
Trainable params: 79,510
Non-trainable params: 0
_________________________________________________________________
```

은닉층 뒤에 드롭아웃 층은 훈련되는 모델 파라미터가 없다. 또한 입력과 출력의 크기가 같다.
일부 뉴런의 출력을 0으로 만들지만, 전체 배열의 크기를 바꾸지는 않는다.

훈련이 끝난 뒤에 평가나 예측을 사용할 때에는 드롭아웃을 적용하지 말아야 한다.
훈련된 모든 뉴런을 사용해야 올바른 예측을 수행할 수 있다 -> 모델 훈련 뒤에는 드롭아웃 층을 다시 빼야 한다.

다행히도 텐서플로와 케라스는 모델을 평가와 예측에 사용할 때는 자동으로 드롭아웃을 적용하지 않는다.
그래서 마음 편하게 검증 점수를 계산할 수 있다. 이전과 마찬가지로
훈련 손실과 검증 손실의 그래프를 그려 보자.

In [33]:
```python
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics='accuracy')
history = model.fit(train_scaled, train_target, epochs=20, verbose=0, validation_data=(val_scaled, val_target))

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

plt.show()
```



열 번째 에포크 정도에서 검증 손실의 감소가 멈추지만 크게 상승하지 않고, 어느 정도 유지되고 있다.

20번의 에포크 동안 훈련을 했기 때문에 결국 다소 과대적합되어 있다.
그렇다면 과대적합되지 않은 모델을 얻기 위해 에포크 횟수를 10으로 하고 다시 훈련해야겠다.

모델 지정과 복원

In [48]:
```python
model = model_fn(keras.layers.Dropout(0.3))
model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy', metrics = 'accuracy')
history = model.fit(train_scaled, train_target, epochs=10, verbose = 0, validation_data = (val_scaled, val_target))
```

케라스 모델은 훈련된 모델의 파라미터를 저장하는 간편한 `save_weights()` 메서드를 제공한다.

기본적으로 이 메서드는 텐서플로의 체크포인트 포맷으로 저장하지만 파일의 확장자가 **'.h5'**일 경우 HDF5 모델로 저장한다.

In [49]:
```python
model.save_weights('model-weights.h5')
```

또한 모델 구조와 모델 파라미터를 함께 저장하는 save() 메서드도 제공한다. 기본적으로 이 메서드는 텐서플로의 SavedModel 포맷으로 저장하지만
파일의 확장자가 '.h5' 일 경우 HDF5 포맷으로 저장한다.
https://wikidocs.net/33122

In [50]:
```python
model.save('model-whole.h5')
```

이 두 파일이 잘 만들어졌는지 확인해 보자.

In [51]:
```python
# google colab 내의 디렉토리에서 확인한 결과
!ls -al *.h5
```

```
-rw-r--r-- 1 root root 982664 Jan 23 07:24 model-total.h5
-rw-r--r-- 1 root root 333448 Jan 23 07:40 model-weights.h5
-rw-r--r-- 1 root root 982664 Jan 23 07:40 model-whole.h5
```

두 가지 실험을 해보자.

1. 훈련을 하지 않은 새로운 모델을 만들고 model-weights.h5 파일에서 훈련된 모델 파라미터를 읽어서 사용
2. 아예 model-whole.h5 파일에서 새로운 모델을 만들어 바로 사용

먼저 첫 번째 실험부터 시작해 보자.

In [52]:
```python
model = model_fn(keras.layers.Dropout(0.3))
model.load_weights('model-weights.h5')
```

훈련하지 않은 새로운 모델을 만들고 이전에 저장했던 모델 파라미터를 적재했다.
이때 사용하는 메서드는 save_weights()와 쌍을 이루는 load_weights() 메서드이다.

이 모델의 검증 정확도를 확인해 보자.
케라스에서 예측을 수행하는 **predict()** 메서드는 사이킷런과 달리
샘플마다 10개의 클래스에 대한 확률을 반환한다. 패션 MNIST 데이터셋이 다중 분류 문제이기 때문이다.

패션 MNIST 데이터셋에서 덜어낸 검증 세트의 샘플 개수는 12,000개이기 때문에 predict() 메서드는
(12000, 10) 크기의 배열을 반환한다.

따라서 조금 번거롭지만 10개 확률 중에 가장 큰 값의 인덱스를 골라 타깃 레이블과 비교하여 정확도를 계산해 보자.

In [53]:
```python
import numpy as np
val_labels = np.argmax(model.predict(val_scaled), axis = -1) # 행 단위 예측
print(np.mean(val_labels == val_target))
```

```
375/375 [==============================] - 1s 2ms/step
0.7031666666666667
```

모델이 predict() 메서드 결과에서 가장 큰 값을 고르기 위해 넘파이 argmax 함수를 사용했다.
이 함수는 배열에서 가장 큰 값의 인덱스를 반환한다.
예를 들어 배열의 첫 번째 원소가 가장 큰 값일 경우 0을 반환한다. 우리가 준비한 타깃값도 0부터 시작하기 때문에 비교하기 좋다.

axis = 1 : 배열의 마지막 차원을 따라 최댓값을 고른다.
검증 세트는 2차원 배열이기 때문에 마지막 차원은 -1이 된다.

argmax()로 고른 인덱스(val_labels)와 타깃(val_target)을 비교한다.
두 배열에서 각 위치의 값이 같으면 1이 되고, 다르면 0이 된다. 이를 평균하면 정확도가 된다.

이번에는 모델 전체를 파일에서 읽은 다음 검증 세트의 정확도를 출력해 보자.
모델이 저장된 파일을 읽을 때는 케라스가 제공하는 `load_model()` 함수를 제공한다.

In [54]:
```python
model = keras.models.load_model('model-whole.h5')

model.evaluate(val_scaled, val_target)
```

Out[54]:
```
375/375 [==============================] - 1s 2ms/step - loss: 0.7782 - accuracy: 0.7032
[0.7781519889831543, 0.703166663646698]
```

같은 모델을 저장하고 다시 불러들였기 때문에 위와 동일한 정확도를 얻었다.

그런데 이 과정을 돌이켜 보면,
20번의 에포크 동안 모델을 훈련하여 검증 점수가 상승하는 지점을 확인했다.
그다음 모델을 과대적합되지 않는 에포크만큼 다시 훈련했다.
모델을 두 번씩 훈련하지 않고 한 번에 끝낼 수는 없을까 ??

콜백

- 이제 케라스의 콜백을 사용할 차례이다.

콜백 :

- 훈련 과정 중간에 어떤 작업을 수행할 수 있게 해주는 객체
- keras.callbacks 패키지 아래 있는 클래스들
- fit() 메서드의 callbacks 매개변수에 리스트로 전달하여 사용한다.

여기서 사용할 ModelCheckpoint 콜백은 기본적으로 에포크마다 모델을 저장한다.

- `save_best_only = True` 매개변수를 지정하여 가장 낮은 검증 점수를 만드는 모델을 저장 가능
- 저장될 파일 이름을 'best-model.h5'로 지정하여 콜백을 적용시켜 보자.

In [55]:
```python
model = model_fn(keras.layers.Dropout(0.3))
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics='accuracy')
checkpoint_cb = keras.callbacks.ModelCheckpoint('best-model.h5', save_best_only = True)
model.fit(train_scaled, train_target, epochs=20, verbose = 0, validation_data=(val_scaled, val_target), callbacks=[checkpoint_cb])
```

Out[55]:
```
<keras.callbacks.History at 0x7f2919124850>
```

ModelCheckpoint 클래스의 객체 checkpoint_cb를 만든 후 fit()메서드의 callbacks 매개변수에 리스트로 감싸서 전달
모델이 훈련한 후에 best-model.h5라는 파일에 최상의 검증 점수를 낸 모델이 저장된다.

이 모델을 load_model() 함수로 다시 읽어서 예측을 수행해 보자.

In [56]:
```python
model = keras.models.load_model('best-model.h5')
model.evaluate(val_scaled, val_target)
```

Out[56]:
```
375/375 [==============================] - 1s 2ms/step - loss: 0.6837 - accuracy: 0.7330
[0.6836659908294678, 0.7329999804496765]
```

ModelCheckpoint 콜백이 가장 낮은 검증 점수의 모델을 자동으로 저장해 주었다.
하지만 여전히 20번의 에포크 동안 훈련한다.
사실 검증 점수가 상승하기 시작하면 이후에는 과대적합이 더 커지기 때문에 훈련을 계속할 필요가 없다.

이렇게 과대적합이 시작되기 전에 훈련을 미리 종료하는 것을 ***조기 종료***라 한다.

규제 방법 중 하나라고 생각할 수도 있다.

케라스에서는 조기 종료를 위한 `EarlyStopping` 콜백을 제공한다.
이 콜백의 `patience` 매개변수는 검증 점수가 향상되지 않더라도 참을 에포크 횟수로 지정한다.
예를 들어 patience = 2라면, 2번 연속 검증 점수가 향상되지 않으면 훈련을 중지한다.

또한 `restore_best_weights = True` 매개변수를 True로 지정하면
가장 낮은 검증 손실을 낸 모델 파라미터로 되돌린다.

EarlyStopping 콜백을 ModelCheckpoint 콜백과 함께 사용하면 가장 낮은 검증 손실의 모델을 파일에 저장하고,
검증 손실이 다시 상승할 때 훈련을 중지할 수 있다.
또한 훈련을 중지한 다음 현재 모델의 파라미터를 최상의 파라미터로 되돌린다.

이 두 콜백을 함께 사용해 보자.

In [57]:
```python
model = model_fn(keras.layers.Dropout(0.3))
model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy', metrics='accuracy')
# -- 여기까지는 동일

# callbacks
checkpoint_cb = keras.callbacks.ModelCheckpoint('best-model.h5', save_best_only = True)
early_stopping_cb = keras.callbacks.EarlyStopping(patience = 2, restore_best_weights = True)

history = model.fit(train_scaled, train_target, epochs = 20, verbose = 0, validation_data = (val_scaled, val_target), callbacks = [checkpoint_cb, early_stopping_cb])
```

EarlyStopping 콜백을 추가한 것 외에는 이전과 동일
fit(), callbacks 매개변수를 2개를 전달한 것에 해당

훈련을 마치고 나면 몇 번째 에포크에서 훈련이 중지되었는지
early_stopping_cb 객체의 stopped_epoch 속성에서 확인할 수 있다.

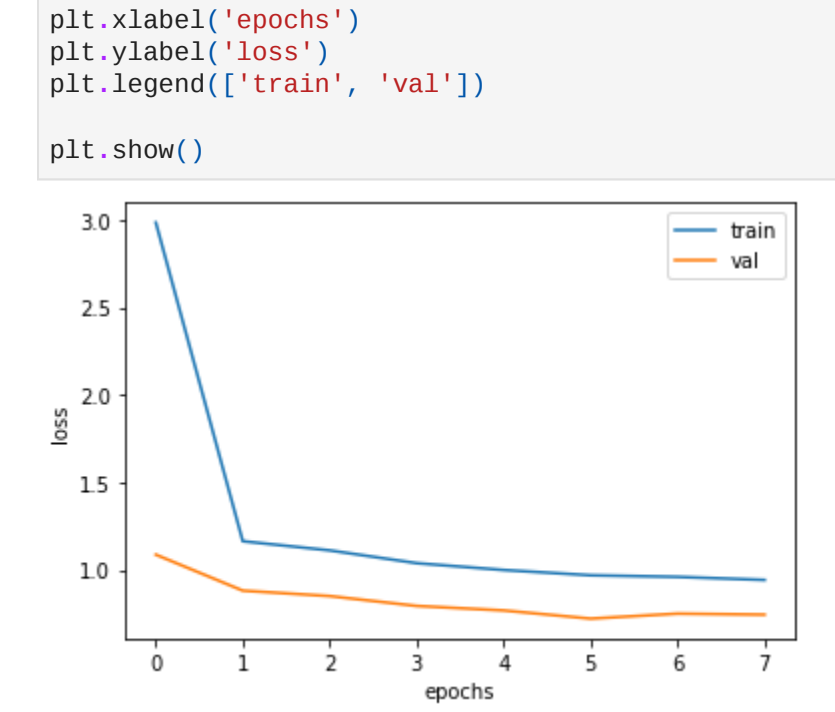
In [59]:
```python
print(early_stopping_cb.stopped_epoch)
```

```
7
```

에포크 횟수가 0부터 시작해서 7번째 : 8번째 에포크에서 종료되었다.
patience를 2로 설정했기 때문에, 최상의 모델은 6번째 epoch일 것이다.

훈련 손실과 검증 손실을 출력해서 확인해 보자.

In [61]:
```python
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.xlabel('epochs')
plt.ylabel('loss')
plt.legend(['train', 'val'])

plt.show()
```



6번째 에포크에서 가장 낮은 손실을 기록했고, 8번째 에포크에서 종료되었다.
조기 종료 기법을 사용하면 안심하고 에포크 횟수를 크게 지정해도 괜찮다.
컴퓨터 자원과 시간을 아낄 수 있고,
ModelCheckpoint 콜백과 함께 사용하면 최상의 모델을 자동으로 저장해 주므로 편리하다.

마지막으로 조기 종료로 얻은 모델을 사용해 검증 세트에 대한 성능을 확인해 보자.

In [62]:
```python
model.evaluate(val_scaled, val_target)
```

Out[62]:
```
375/375 [==============================] - 1s 3ms/step - loss: 0.7209 - accuracy: 0.7230
[0.7208757400512695, 0.7229999899864197]
```