뉴스 기사 분류: 다중 분류 문제

학습 내용

- 01 로이터 뉴스를 이용한 신경망 구현
- 02 다항분류 문제
  - 출력 클래스가 2에서 46개의 클래스로 구분

In [35]:

```
import keras
keras.__version__
```

Out[35]:

```
'2.4.3'
```

로이터 뉴스를 46개의 상호 배타적인 토픽으로 분류하는 신경망

- 1986년에 로이터에서 공개한 짧은 뉴스 기사와 토픽의 집합인 로이터 데이터셋을 사용
- 46개의 토픽
- 각 토픽은 훈련 세트에 최소한 10개의 샘플

In [36]:

```
from keras.datasets import reuters

(train_data, train_labels), (test_data, test_labels) = reuters.load_data(num_words=10000)
```

```
c:₩users₩toto₩anaconda3₩envs₩tf2x₩lib₩site-packages₩tensorflow₩python₩keras₩datasets
₩reuters.py:148: VisibleDeprecationWarning: Creating an ndarray from ragged nested s
equences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different len
gths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=obje
ct' when creating the ndarray
  x_train, y_train = np.array(xs[:idx]), np.array(labels[:idx])
c:₩users₩toto₩anaconda3₩envs₩tf2x₩lib₩site-packages₩tensorflow₩python₩keras₩datasets
₩reuters.py:149: VisibleDeprecationWarning: Creating an ndarray from ragged nested s
equences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different len
gths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=obje
ct' when creating the ndarray
  x_test, y_test = np.array(xs[idx:]), np.array(labels[idx:])
```

- IMDB 데이터셋에서처럼 num_words=10000 매개변수는 데이터에서 가장 자주 등장하는 단어 10,000개로 제한

In [37]:

```
len(train_data), len(test_data), len(train_data)+ len(test_data)
```

Out[37]:

```
(8982, 2246, 11228)
```

In [38]:

```
print(train_labels)
```

```
[ 3  4  3 ... 25  3 25]
```

In [39]:

```
train_data[10][0:15]
```

Out[39]:

```
[1, 245, 273, 207, 156, 53, 74, 160, 26, 14, 46, 296, 26, 39, 74]
```

단어로 디코딩

In [40]:

```
dir(reuters)
```

Out[40]:

```
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'get_word_index',
 'load_data']
```

In [41]:

```
word_index = reuters.get_word_index()
reverse_word_index = dict([(value, key)
                           for (key, value) in word_index.items()])

# 0, 1, 2는 '패딩', '문서 시작', '사전에 없음'을 위한 인덱스이므로 3을 뺍니다
decoded_newswire = ' '.join([reverse_word_index.get(i - 3, '?')
                             for i in train_data[0]])
```

In [42]:

```
decoded_newswire
```

Out[42]:

```
'? ? ? said as a result of its december acquisition of space co it expects earnings
per share in 1987 of 1 15 to 1 30 dlrs per share up from 70 cts in 1986 the company
said pretax net should rise to nine to 10 mln dlrs from six mln dlrs in 1986 and ren
tal operation revenues to 19 to 22 mln dlrs from 12 5 mln dlrs it said cash flow per
share this year should be 2 50 to three dlrs reuter 3'
```

샘플과 연결된 레이블

- 토픽의 인덱스로 0과 45사이의 정수

In [43]:

```
train_labels[0:10]
```

Out[43]:

```
array([ 3,  4,  3,  4,  4,  4,  4,  3,  3, 16], dtype=int64)
```

데이터 준비

In [44]:

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

# 훈련 데이터 벡터 변환
X_train = vectorize_sequences(train_data)

# 테스트 데이터 벡터 변환
X_test = vectorize_sequences(test_data)

print("변환 전 : ", train_data.shape, test_data.shape)
print("변환 후 : ", X_train.shape, X_test.shape)
```

```
변환 전 :  (8982,) (2246,)
변환 후 :  (8982, 10000) (2246, 10000)
```

레이블을 벡터로 바꾸는 방법은 두 가지

- 레이블의 리스트를 정수 텐서로 변환하는 것과 원-핫 인코딩을 사용하는 것

In [45]:

```
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results

# 출력(레이블)을 벡터 변환(원핫)
# 훈련 레이블 벡터 변환
one_hot_train_labels = to_one_hot(train_labels)
# 테스트 레이블 벡터 변환
one_hot_test_labels = to_one_hot(test_labels)
```

- MNIST 예제에서 이미 보았듯이 케라스에는 이를 위한 내장 함수

모델 구성

- 마지막 출력이 46차원이기 때문에 중간층의 히든 유닛이 46개보다 많이 적어서는 안된다.
- 마지막 Dense 층의 크기가 46 : 각 입력 샘플에 대해서 46차원의 벡터를 출력
- 마지막 층에 softmax 활성화 함수을 사용
- 각 입력 샘플마다 46개의 출력 클래스에 대한 확률 분포를 출력
- 즉, 46차원의 출력 벡터를 만들며 output[i]는 어떤 샘플이 클래스 i에 속할 확률입니다. 46개의 값을 모두 더하면 1이 됩니다.
- 손실 함수는 categorical_crossentropy
  - 이 함수는 두 확률 분포의 사이의 거리를 측정
  - 네트워크가 출력한 확률 분포와 진짜 레이블의 분포 사이의 거리
    - 두 분포 사이의 거리를 최소화하면 진짜 레이블에 가능한 가까운 출력을 내도록 모델을 훈련

In [46]:

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

In [47]:

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

모델 검증

- 훈련 데이터에서 8,892개에서 1,000개의 샘플을 따로 떼어서 검증 세트로 사용

In [48]:

```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]

y_val = one_hot_train_labels[:1000]
partial_y_train = one_hot_train_labels[1000:]
```

In [49]:

```
# 4. 모델 학습시키기
from keras.callbacks import EarlyStopping
early_stopping = EarlyStopping(patience = 20) # 조기종료 콜백함수 정의
```

In [50]:

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=50,
                    batch_size=512,
                    validation_data=(x_val, y_val),
                    callbacks=[early_stopping])
```

```
Epoch 1/50
16/16 [==============================] - 3s 84ms/step - loss: 3.1371 - accuracy: 0.3
935 - val_loss: 1.8008 - val_accuracy: 0.6390
Epoch 2/50
16/16 [==============================] - 1s 41ms/step - loss: 1.5332 - accuracy: 0.6
936 - val_loss: 1.3366 - val_accuracy: 0.7150
Epoch 3/50
16/16 [==============================] - 1s 45ms/step - loss: 1.1073 - accuracy: 0.7
509 - val_loss: 1.1616 - val_accuracy: 0.7500
Epoch 4/50
16/16 [==============================] - 1s 44ms/step - loss: 0.8715 - accuracy: 0.8
124 - val_loss: 1.0752 - val_accuracy: 0.7670
Epoch 5/50
16/16 [==============================] - 1s 40ms/step - loss: 0.7034 - accuracy: 0.8
469 - val_loss: 0.9822 - val_accuracy: 0.8040
Epoch 6/50
16/16 [==============================] - 1s 41ms/step - loss: 0.5584 - accuracy: 0.8
864 - val_loss: 0.9381 - val_accuracy: 0.8060
Epoch 7/50
16/16 [==============================] - 1s 40ms/step - loss: 0.4362 - accuracy: 0.9
126 - val_loss: 0.9529 - val_accuracy: 0.7940
Epoch 8/50
16/16 [==============================] - 1s 41ms/step - loss: 0.3546 - accuracy: 0.9
273 - val_loss: 0.9001 - val_accuracy: 0.8160
Epoch 9/50
16/16 [==============================] - 1s 40ms/step - loss: 0.2812 - accuracy: 0.9
416 - val_loss: 0.9181 - val_accuracy: 0.8120
Epoch 10/50
16/16 [==============================] - 1s 39ms/step - loss: 0.2381 - accuracy: 0.9
466 - val_loss: 0.9355 - val_accuracy: 0.8190
Epoch 11/50
16/16 [==============================] - 1s 40ms/step - loss: 0.2126 - accuracy: 0.9
507 - val_loss: 0.9208 - val_accuracy: 0.8180
Epoch 12/50
16/16 [==============================] - 1s 41ms/step - loss: 0.1827 - accuracy: 0.9
567 - val_loss: 0.9273 - val_accuracy: 0.8200
Epoch 13/50
16/16 [==============================] - 1s 41ms/step - loss: 0.1465 - accuracy: 0.9
609 - val_loss: 0.9583 - val_accuracy: 0.8120
Epoch 14/50
16/16 [==============================] - 1s 41ms/step - loss: 0.1352 - accuracy: 0.9
606 - val_loss: 1.0075 - val_accuracy: 0.8040
Epoch 15/50
16/16 [==============================] - 1s 41ms/step - loss: 0.1309 - accuracy: 0.9
580 - val_loss: 1.0189 - val_accuracy: 0.8110
Epoch 16/50
16/16 [==============================] - 1s 40ms/step - loss: 0.1285 - accuracy: 0.9
581 - val_loss: 1.0068 - val_accuracy: 0.8150
Epoch 17/50
16/16 [==============================] - 1s 41ms/step - loss: 0.1223 - accuracy: 0.9
576 - val_loss: 1.0643 - val_accuracy: 0.8000
```

```
Epoch 18/50
16/16 [==============================] - 1s 40ms/step - loss: 0.1066 - accuracy: 0.9
619 - val_loss: 1.0524 - val_accuracy: 0.8120
Epoch 19/50
16/16 [==============================] - 1s 42ms/step - loss: 0.1036 - accuracy: 0.9
607 - val_loss: 1.1199 - val_accuracy: 0.7960
Epoch 20/50
16/16 [==============================] - 1s 43ms/step - loss: 0.0970 - accuracy: 0.9
648 - val_loss: 1.0609 - val_accuracy: 0.8070
Epoch 21/50
16/16 [==============================] - 1s 43ms/step - loss: 0.1035 - accuracy: 0.9
609 - val_loss: 1.1085 - val_accuracy: 0.8030
Epoch 22/50
16/16 [==============================] - 1s 40ms/step - loss: 0.0975 - accuracy: 0.9
635 - val_loss: 1.0940 - val_accuracy: 0.8020
Epoch 23/50
16/16 [==============================] - 1s 44ms/step - loss: 0.0876 - accuracy: 0.9
632 - val_loss: 1.1190 - val_accuracy: 0.8000
Epoch 24/50
16/16 [==============================] - 1s 41ms/step - loss: 0.0907 - accuracy: 0.9
626 - val_loss: 1.2240 - val_accuracy: 0.7880
Epoch 25/50
16/16 [==============================] - 1s 41ms/step - loss: 0.0907 - accuracy: 0.9
638 - val_loss: 1.2303 - val_accuracy: 0.7920
Epoch 26/50
16/16 [==============================] - 1s 40ms/step - loss: 0.0868 - accuracy: 0.9
608 - val_loss: 1.1235 - val_accuracy: 0.8010
Epoch 27/50
16/16 [==============================] - 1s 39ms/step - loss: 0.0866 - accuracy: 0.9
633 - val_loss: 1.2316 - val_accuracy: 0.7910
Epoch 28/50
16/16 [==============================] - 1s 39ms/step - loss: 0.0878 - accuracy: 0.9
617 - val_loss: 1.1501 - val_accuracy: 0.7990
```

손실과 정확도 곡선

In [51]:

```
import matplotlib.pyplot as plt
```

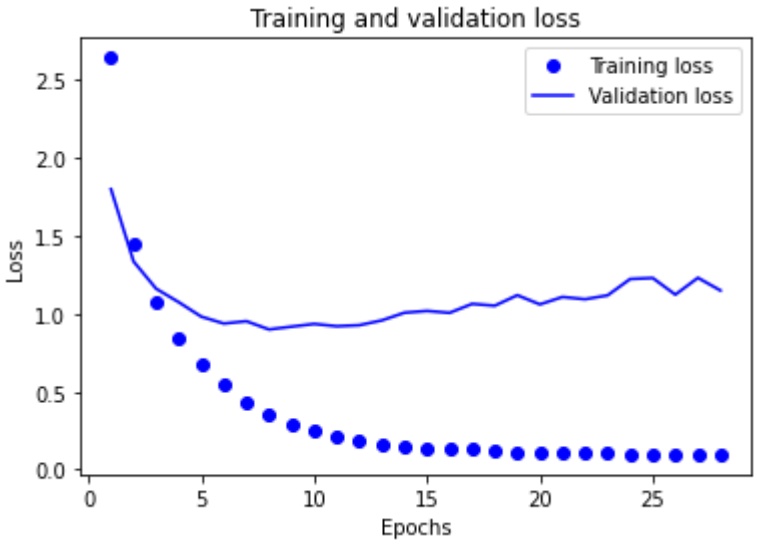
In [52]:

```
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



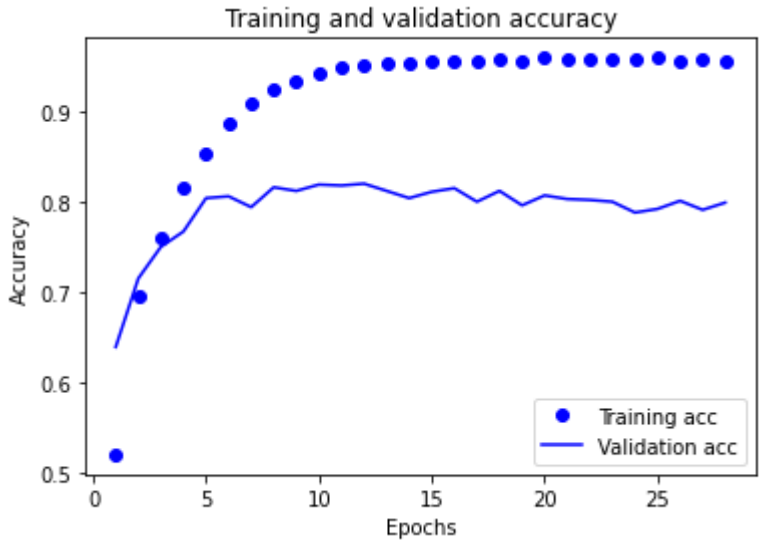
In [53]:

```
plt.clf()   # 그래프를 초기화합니다

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



9번째 에포크 이후에 과대적합 시작. 9번의 에포크로 새로운 모델 훈련과 테스트 세트에서 평가

In [54]:

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

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

model.fit(partial_x_train,
          partial_y_train,
          epochs=9,
          batch_size=512,
          validation_data=(x_val, y_val))

results = model.evaluate(x_test, one_hot_test_labels)
```

```
Epoch 1/9
16/16 [==============================] - 3s 70ms/step - loss: 3.0570 - accuracy: 0.3
573 - val_loss: 1.6884 - val_accuracy: 0.6550
Epoch 2/9
16/16 [==============================] - 1s 41ms/step - loss: 1.4641 - accuracy: 0.6
951 - val_loss: 1.2945 - val_accuracy: 0.7260
Epoch 3/9
16/16 [==============================] - 1s 41ms/step - loss: 1.0594 - accuracy: 0.7
721 - val_loss: 1.1276 - val_accuracy: 0.7540
Epoch 4/9
16/16 [==============================] - 1s 39ms/step - loss: 0.8001 - accuracy: 0.8
335 - val_loss: 1.0186 - val_accuracy: 0.7900
Epoch 5/9
16/16 [==============================] - 1s 39ms/step - loss: 0.6623 - accuracy: 0.8
680 - val_loss: 0.9695 - val_accuracy: 0.7950
Epoch 6/9
16/16 [==============================] - 1s 40ms/step - loss: 0.5055 - accuracy: 0.8
987 - val_loss: 0.9150 - val_accuracy: 0.8110
Epoch 7/9
16/16 [==============================] - 1s 40ms/step - loss: 0.4065 - accuracy: 0.9
222 - val_loss: 0.9219 - val_accuracy: 0.8020
Epoch 8/9
16/16 [==============================] - 1s 41ms/step - loss: 0.3171 - accuracy: 0.9
331 - val_loss: 0.9093 - val_accuracy: 0.8090
Epoch 9/9
16/16 [==============================] - 1s 39ms/step - loss: 0.2743 - accuracy: 0.9
396 - val_loss: 0.8979 - val_accuracy: 0.8140
71/71 [==============================] - 0s 3ms/step - loss: 0.9667 - accuracy: 0.79
21
```

In [55]:

```
results
```

Out[55]:

```
[0.966707170009613, 0.7920747995376587]
```

In [56]:

```
import copy

test_labels_copy = copy.copy(test_labels)
np.random.shuffle(test_labels_copy)
float(np.sum(np.array(test_labels) == np.array(test_labels_copy))) / len(test_labels)
```

Out[56]:

0.18744434550311664

새로운 데이터로 예측

In [57]:

```
predictions = model.predict(x_test)
```

In [58]:

```
predictions[0].shape
```

Out[58]:

(46,)

In [59]:

```
np.sum(predictions[0])
```

Out[59]:

0.9999998

가장 큰 값이 예측 클래스가 된다.

In [60]:

```
np.argmax(predictions[0])
```

Out[60]:

3

정수 레이블을 그대로 사용할 때

- 손실함수를 sparse_categorical_crossentropy 를 사용

In [63]:

```
y_train = np.array(train_labels)
y_test = np.array(test_labels)

print(y_train.shape)
```

```
(8982,)
```

In [64]:

```
model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy', metrics=['acc'])
```

In [65]:

```
x_val = x_train[ :1000]
partial_x_train = x_train[1000:]

y_val = y_train[:1000]
partial_y_train = y_train[1000:]
```

In [66]:

```
partial_x_train.shape, partial_y_train.shape
```

Out[66]:

```
((7982, 10000), (7982,))
```

In [67]:

```
## 학습을 진행
history = model.fit(partial_x_train, partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

```
Epoch 1/20
16/16 [==============================] - 3s 78ms/step - loss: 0.2591 - acc: 0.9351 -
val_loss: 0.9058 - val_acc: 0.8220
Epoch 2/20
16/16 [==============================] - 1s 46ms/step - loss: 0.1782 - acc: 0.9563 -
val_loss: 0.9286 - val_acc: 0.8160
Epoch 3/20
16/16 [==============================] - 1s 49ms/step - loss: 0.1607 - acc: 0.9546 -
val_loss: 0.9281 - val_acc: 0.8140
Epoch 4/20
16/16 [==============================] - 1s 46ms/step - loss: 0.1394 - acc: 0.9589 -
val_loss: 0.9835 - val_acc: 0.8150
Epoch 5/20
16/16 [==============================] - 1s 48ms/step - loss: 0.1351 - acc: 0.9597 -
val_loss: 0.9883 - val_acc: 0.8210
Epoch 6/20
16/16 [==============================] - 1s 47ms/step - loss: 0.1328 - acc: 0.9568 -
val_loss: 1.0479 - val_acc: 0.8010
Epoch 7/20
16/16 [==============================] - 1s 47ms/step - loss: 0.1137 - acc: 0.9607 -
val_loss: 1.0699 - val_acc: 0.8020
Epoch 8/20
16/16 [==============================] - 1s 49ms/step - loss: 0.1159 - acc: 0.9625 -
val_loss: 1.0392 - val_acc: 0.8030
Epoch 9/20
16/16 [==============================] - 1s 48ms/step - loss: 0.1036 - acc: 0.9638 -
val_loss: 1.0536 - val_acc: 0.8100
Epoch 10/20
16/16 [==============================] - 1s 49ms/step - loss: 0.1058 - acc: 0.9618 -
val_loss: 1.0603 - val_acc: 0.8140
Epoch 11/20
16/16 [==============================] - 1s 48ms/step - loss: 0.0973 - acc: 0.9626 -
val_loss: 1.0696 - val_acc: 0.8150
Epoch 12/20
16/16 [==============================] - 1s 47ms/step - loss: 0.0984 - acc: 0.9627 -
val_loss: 1.1199 - val_acc: 0.8080
Epoch 13/20
16/16 [==============================] - 1s 49ms/step - loss: 0.0927 - acc: 0.9633 -
val_loss: 1.1388 - val_acc: 0.8030
Epoch 14/20
16/16 [==============================] - 1s 49ms/step - loss: 0.0954 - acc: 0.9624 -
val_loss: 1.1131 - val_acc: 0.8090
Epoch 15/20
16/16 [==============================] - 1s 37ms/step - loss: 0.0947 - acc: 0.9572 -
val_loss: 1.1343 - val_acc: 0.8090
Epoch 16/20
16/16 [==============================] - 1s 37ms/step - loss: 0.0893 - acc: 0.9650 -
val_loss: 1.2032 - val_acc: 0.7970
Epoch 17/20
16/16 [==============================] - 1s 36ms/step - loss: 0.0861 - acc: 0.9635 -
val_loss: 1.1824 - val_acc: 0.7960
Epoch 18/20
```

```
16/16 [==============================] - 1s 36ms/step - loss: 0.0832 - acc: 0.9646 -
val_loss: 1.1536 - val_acc: 0.8100
Epoch 19/20
16/16 [==============================] - 1s 36ms/step - loss: 0.0903 - acc: 0.9616 -
val_loss: 1.1679 - val_acc: 0.8060
Epoch 20/20
16/16 [==============================] - 1s 35ms/step - loss: 0.0859 - acc: 0.9638 -
val_loss: 1.2039 - val_acc: 0.8000
```

충분히 큰 중간층을 두기

- 출력층이 46차원이다. 중간층의 히든 유닛이 46개보다 적으면 안된다.

In [68]:

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

In [69]:

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

In [70]:

```
partial_x_train.shape, partial_y_train.shape
```

Out[70]:

```
((7982, 10000), (7982,))
```

In [71]:

```
history = model.fit(partial_x_train, partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

```
Epoch 1/20
16/16 [==============================] - 2s 64ms/step - loss: 3.1455 - accuracy: 0.3
767 - val_loss: 1.7587 - val_accuracy: 0.5990
Epoch 2/20
16/16 [==============================] - 1s 39ms/step - loss: 1.5414 - accuracy: 0.6
722 - val_loss: 1.3181 - val_accuracy: 0.7030
Epoch 3/20
16/16 [==============================] - 1s 40ms/step - loss: 1.1197 - accuracy: 0.7
631 - val_loss: 1.1503 - val_accuracy: 0.7490
Epoch 4/20
16/16 [==============================] - 1s 40ms/step - loss: 0.8629 - accuracy: 0.8
194 - val_loss: 1.0611 - val_accuracy: 0.7830
Epoch 5/20
16/16 [==============================] - 1s 40ms/step - loss: 0.6675 - accuracy: 0.8
668 - val_loss: 0.9864 - val_accuracy: 0.7990
Epoch 6/20
16/16 [==============================] - 1s 37ms/step - loss: 0.5050 - accuracy: 0.8
988 - val_loss: 0.9277 - val_accuracy: 0.8150
Epoch 7/20
16/16 [==============================] - 1s 40ms/step - loss: 0.4284 - accuracy: 0.9
128 - val_loss: 0.9207 - val_accuracy: 0.8100
Epoch 8/20
16/16 [==============================] - 1s 40ms/step - loss: 0.3431 - accuracy: 0.9
302 - val_loss: 0.9010 - val_accuracy: 0.8150
Epoch 9/20
16/16 [==============================] - 1s 39ms/step - loss: 0.2889 - accuracy: 0.9
388 - val_loss: 0.9275 - val_accuracy: 0.8090
Epoch 10/20
16/16 [==============================] - 1s 40ms/step - loss: 0.2351 - accuracy: 0.9
460 - val_loss: 0.9145 - val_accuracy: 0.8220
Epoch 11/20
16/16 [==============================] - 1s 40ms/step - loss: 0.2114 - accuracy: 0.9
486 - val_loss: 1.0359 - val_accuracy: 0.7840
Epoch 12/20
16/16 [==============================] - 1s 41ms/step - loss: 0.1884 - accuracy: 0.9
552 - val_loss: 0.9701 - val_accuracy: 0.8120
Epoch 13/20
16/16 [==============================] - 1s 41ms/step - loss: 0.1635 - accuracy: 0.9
556 - val_loss: 1.0116 - val_accuracy: 0.8110
Epoch 14/20
16/16 [==============================] - 1s 38ms/step - loss: 0.1313 - accuracy: 0.9
610 - val_loss: 0.9999 - val_accuracy: 0.8190
Epoch 15/20
16/16 [==============================] - 1s 39ms/step - loss: 0.1324 - accuracy: 0.9
611 - val_loss: 0.9757 - val_accuracy: 0.8080
Epoch 16/20
16/16 [==============================] - 1s 37ms/step - loss: 0.1192 - accuracy: 0.9
607 - val_loss: 1.0974 - val_accuracy: 0.8060
Epoch 17/20
16/16 [==============================] - 1s 40ms/step - loss: 0.1217 - accuracy: 0.9
624 - val_loss: 1.0700 - val_accuracy: 0.8080
Epoch 18/20
16/16 [==============================] - 1s 40ms/step - loss: 0.1071 - accuracy: 0.9
```

```
631 - val_loss: 1.0466 - val_accuracy: 0.8140
Epoch 19/20
16/16 [==============================] - 1s 40ms/step - loss: 0.1059 - accuracy: 0.9
617 - val_loss: 1.1084 - val_accuracy: 0.8070
Epoch 20/20
16/16 [==============================] - 1s 37ms/step - loss: 0.0991 - accuracy: 0.9
644 - val_loss: 1.1524 - val_accuracy: 0.7860
```

46차원보다 훨씬 작은 중간층(예를 들어 4차원)을 두면,

In [72]:

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

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

history = model.fit(partial_x_train, partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

```
Epoch 1/20
16/16 [==============================] - 3s 100ms/step - loss: 3.4718 - accuracy: 0.
2483 - val_loss: 2.6496 - val_accuracy: 0.4840
Epoch 2/20
16/16 [==============================] - 1s 45ms/step - loss: 2.4698 - accuracy: 0.5
439 - val_loss: 2.1270 - val_accuracy: 0.5650
Epoch 3/20
16/16 [==============================] - 1s 42ms/step - loss: 2.0090 - accuracy: 0.5
775 - val_loss: 1.8436 - val_accuracy: 0.5840
Epoch 4/20
16/16 [==============================] - 1s 40ms/step - loss: 1.7210 - accuracy: 0.6
146 - val_loss: 1.6735 - val_accuracy: 0.6070
Epoch 5/20
16/16 [==============================] - 1s 41ms/step - loss: 1.5328 - accuracy: 0.6
472 - val_loss: 1.5797 - val_accuracy: 0.6180
Epoch 6/20
16/16 [==============================] - 1s 41ms/step - loss: 1.3907 - accuracy: 0.6
626 - val_loss: 1.5106 - val_accuracy: 0.6300
Epoch 7/20
16/16 [==============================] - 1s 37ms/step - loss: 1.2800 - accuracy: 0.6
782 - val_loss: 1.4800 - val_accuracy: 0.6290
Epoch 8/20
16/16 [==============================] - 1s 38ms/step - loss: 1.2112 - accuracy: 0.6
762 - val_loss: 1.4465 - val_accuracy: 0.6310
Epoch 9/20
16/16 [==============================] - 1s 38ms/step - loss: 1.1329 - accuracy: 0.6
976 - val_loss: 1.3982 - val_accuracy: 0.6550
Epoch 10/20
16/16 [==============================] - 1s 37ms/step - loss: 1.1230 - accuracy: 0.7
128 - val_loss: 1.3873 - val_accuracy: 0.6560
Epoch 11/20
16/16 [==============================] - 1s 37ms/step - loss: 1.0141 - accuracy: 0.7
278 - val_loss: 1.3741 - val_accuracy: 0.6680
Epoch 12/20
16/16 [==============================] - 1s 38ms/step - loss: 1.0224 - accuracy: 0.7
277 - val_loss: 1.3788 - val_accuracy: 0.6710
Epoch 13/20
16/16 [==============================] - 1s 37ms/step - loss: 0.9730 - accuracy: 0.7
321 - val_loss: 1.3865 - val_accuracy: 0.6720
Epoch 14/20
16/16 [==============================] - 1s 37ms/step - loss: 0.9322 - accuracy: 0.7
384 - val_loss: 1.3814 - val_accuracy: 0.6730
Epoch 15/20
16/16 [==============================] - 1s 38ms/step - loss: 0.8750 - accuracy: 0.7
```

```
527 - val_loss: 1.3525 - val_accuracy: 0.6780
Epoch 16/20
16/16 [==============================] - 1s 38ms/step - loss: 0.8587 - accuracy: 0.7
587 - val_loss: 1.3522 - val_accuracy: 0.6830
Epoch 17/20
16/16 [==============================] - 1s 38ms/step - loss: 0.8132 - accuracy: 0.7
688 - val_loss: 1.3863 - val_accuracy: 0.6740
Epoch 18/20
16/16 [==============================] - 1s 37ms/step - loss: 0.7769 - accuracy: 0.7
713 - val_loss: 1.3564 - val_accuracy: 0.6820
Epoch 19/20
16/16 [==============================] - 1s 37ms/step - loss: 0.7607 - accuracy: 0.7
762 - val_loss: 1.3677 - val_accuracy: 0.6810
Epoch 20/20
16/16 [==============================] - 1s 36ms/step - loss: 0.7292 - accuracy: 0.7
806 - val_loss: 1.3788 - val_accuracy: 0.6800
```

검증 정확도가 감소

- 검증 정확도의 약간 감소

- 추가 실험해보기
  - 더 크거나 작은 층을 사용해 보세요: 32개 유닛, 128개 유닛 등
  - 여기에서 두 개의 은닉층을 사용했습니다. 한 개의 은닉층이나 세 개의 은닉층을 사용해 보세요.

Summary

- 단일 레이블, 다중 분류 문제에서는 N개의 클래스에 대한 확률 분포를 출력하기 위해 softmax 활성화 함수를 사용
- 항상 범주형 크로스엔트로피를 사용
  - 이 함수는 모델이 출력한 확률 분포와 타깃 분포 사이의 거리를 최소화
- 다중 분류에서 레이블을 다루는 두가지 방법
  - 레이블을 범주형 인코딩(또는 원-핫 인코딩)으로 인코딩하고 categorical_crossentropy 손실 함수를 사용
  - 레이블을 정수로 인코딩하고 sparse_categorical_crossentropy 손실 함수를 사용