뉴스 기사 분류: 다중 분류 문제

학습 내용

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

In [1]:

```
import keras
keras.__version__
```

Out[1]:

```
'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 [2]:

```
from keras.datasets import reuters

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

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/re
uters.npz (https://storage.googleapis.com/tensorflow/tf-keras-datasets/reuters.npz)
2113536/2110848 [==============================] - 1s 0us/step


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 [3]:

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

Out[3]:

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

In [4]:

```
print(train_labels)
```

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

In [5]:

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

Out[5]:

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

단어로 디코딩

In [6]:

```
dir(reuters)
```

Out[6]:

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

In [7]:

```
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]])
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/re
uters_word_index.json (https://storage.googleapis.com/tensorflow/tf-keras-datasets/r
euters_word_index.json)
557056/550378 [==============================] - 0s 0us/step
```

In [8]:

```
decoded_newswire
```

Out[8]:

```
'? ? ? 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 [9]:

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

Out[9]:

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

데이터 준비

In [10]:

```
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 [11]:

```
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)

print("변환 전 : ", train_labels.shape, test_labels.shape)
print("변환 후 : ", one_hot_train_labels.shape, one_hot_test_labels.shape)
```

```
변환 전 :  (8982,) (2246,)
변환 후 :  (8982, 46) (2246, 46)
```

- 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 함수**을 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 [12]:

```
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 [13]:

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

모델 검증

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

In [16]:

```
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 [17]:

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

In [19]:

```
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 [==============================] - 2s 72ms/step - loss: 3.1314 - accuracy: 0.4
078 - val_loss: 1.7646 - val_accuracy: 0.6270
Epoch 2/50
16/16 [==============================] - 1s 38ms/step - loss: 1.4930 - accuracy: 0.6
997 - val_loss: 1.3301 - val_accuracy: 0.7130
Epoch 3/50
16/16 [==============================] - 1s 37ms/step - loss: 1.0791 - accuracy: 0.7
810 - val_loss: 1.1475 - val_accuracy: 0.7490
Epoch 4/50
16/16 [==============================] - 1s 38ms/step - loss: 0.8764 - accuracy: 0.8
122 - val_loss: 1.0385 - val_accuracy: 0.7790
Epoch 5/50
16/16 [==============================] - 1s 39ms/step - loss: 0.6767 - accuracy: 0.8
569 - val_loss: 0.9666 - val_accuracy: 0.8030
Epoch 6/50
16/16 [==============================] - 1s 35ms/step - loss: 0.5501 - accuracy: 0.8
883 - val_loss: 0.9634 - val_accuracy: 0.7900
Epoch 7/50
16/16 [==============================] - 1s 40ms/step - loss: 0.4619 - accuracy: 0.9
068 - val_loss: 0.9213 - val_accuracy: 0.8040
Epoch 8/50
16/16 [==============================] - 1s 38ms/step - loss: 0.3692 - accuracy: 0.9
266 - val_loss: 0.8752 - val_accuracy: 0.8180
Epoch 9/50
16/16 [==============================] - 1s 38ms/step - loss: 0.2830 - accuracy: 0.9
418 - val_loss: 0.8738 - val_accuracy: 0.8240
Epoch 10/50
16/16 [==============================] - 1s 39ms/step - loss: 0.2321 - accuracy: 0.9
497 - val_loss: 0.8939 - val_accuracy: 0.8180
Epoch 11/50
16/16 [==============================] - 1s 36ms/step - loss: 0.2060 - accuracy: 0.9
527 - val_loss: 0.8981 - val_accuracy: 0.8160
Epoch 12/50
16/16 [==============================] - 1s 38ms/step - loss: 0.1752 - accuracy: 0.9
554 - val_loss: 0.9054 - val_accuracy: 0.8110
Epoch 13/50
16/16 [==============================] - 1s 37ms/step - loss: 0.1579 - accuracy: 0.9
596 - val_loss: 0.9604 - val_accuracy: 0.8080
Epoch 14/50
16/16 [==============================] - 1s 38ms/step - loss: 0.1470 - accuracy: 0.9
584 - val_loss: 0.9232 - val_accuracy: 0.8170
Epoch 15/50
16/16 [==============================] - 1s 40ms/step - loss: 0.1376 - accuracy: 0.9
576 - val_loss: 0.9396 - val_accuracy: 0.8160
Epoch 16/50
16/16 [==============================] - 1s 39ms/step - loss: 0.1242 - accuracy: 0.9
589 - val_loss: 0.9662 - val_accuracy: 0.8180
Epoch 17/50
16/16 [==============================] - 1s 38ms/step - loss: 0.1111 - accuracy: 0.9
638 - val_loss: 1.0063 - val_accuracy: 0.8060
```

```
Epoch 18/50
16/16 [==============================] - 1s 35ms/step - loss: 0.1072 - accuracy: 0.9
621 - val_loss: 0.9960 - val_accuracy: 0.8120
Epoch 19/50
16/16 [==============================] - 1s 38ms/step - loss: 0.1034 - accuracy: 0.9
616 - val_loss: 1.0011 - val_accuracy: 0.8090
Epoch 20/50
16/16 [==============================] - 1s 41ms/step - loss: 0.0991 - accuracy: 0.9
610 - val_loss: 1.0502 - val_accuracy: 0.7990
Epoch 21/50
16/16 [==============================] - 1s 38ms/step - loss: 0.0976 - accuracy: 0.9
608 - val_loss: 1.0453 - val_accuracy: 0.8100
Epoch 22/50
16/16 [==============================] - 1s 40ms/step - loss: 0.0926 - accuracy: 0.9
647 - val_loss: 1.1459 - val_accuracy: 0.7860
Epoch 23/50
16/16 [==============================] - 1s 40ms/step - loss: 0.0985 - accuracy: 0.9
627 - val_loss: 1.0846 - val_accuracy: 0.8050
Epoch 24/50
16/16 [==============================] - 1s 36ms/step - loss: 0.0884 - accuracy: 0.9
644 - val_loss: 1.1031 - val_accuracy: 0.7980
Epoch 25/50
16/16 [==============================] - 1s 39ms/step - loss: 0.0875 - accuracy: 0.9
588 - val_loss: 1.1305 - val_accuracy: 0.8010
Epoch 26/50
16/16 [==============================] - 1s 38ms/step - loss: 0.0843 - accuracy: 0.9
632 - val_loss: 1.1108 - val_accuracy: 0.8010
Epoch 27/50
16/16 [==============================] - 1s 37ms/step - loss: 0.0914 - accuracy: 0.9
598 - val_loss: 1.1657 - val_accuracy: 0.7950
Epoch 28/50
16/16 [==============================] - 1s 39ms/step - loss: 0.0841 - accuracy: 0.9
619 - val_loss: 1.1113 - val_accuracy: 0.8080
Epoch 29/50
16/16 [==============================] - 1s 38ms/step - loss: 0.0869 - accuracy: 0.9
612 - val_loss: 1.1391 - val_accuracy: 0.8060
```

손실과 정확도 곡선

In [20]:

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

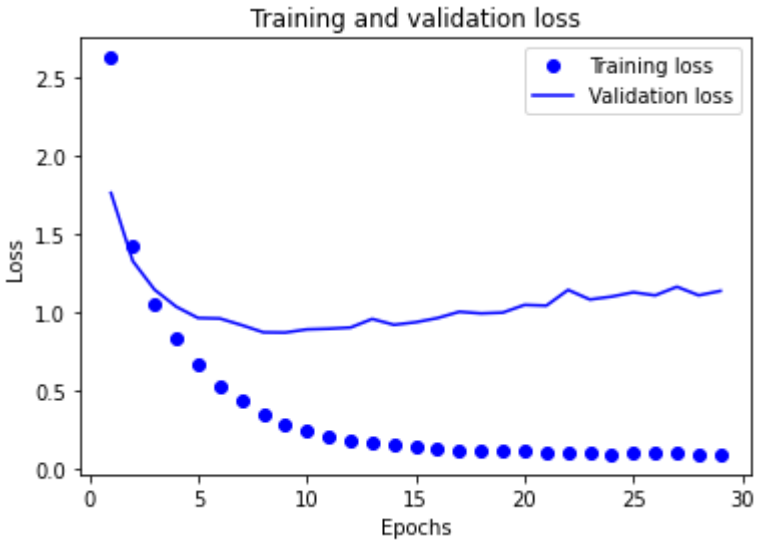
In [21]:

```
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()
```



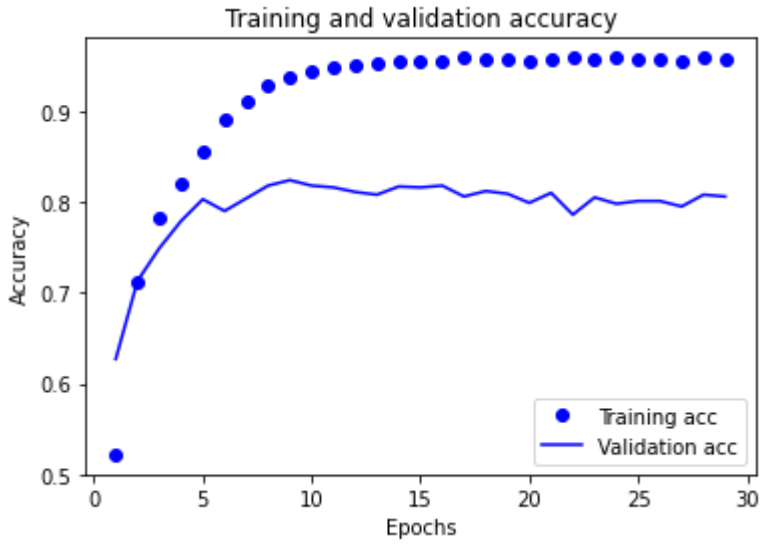
In [22]:

```
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 [24]:

```
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 [==============================] - 2s 61ms/step - loss: 3.2019 - accuracy: 0.3
948 - val_loss: 1.8184 - val_accuracy: 0.6080
Epoch 2/9
16/16 [==============================] - 1s 36ms/step - loss: 1.5925 - accuracy: 0.6
770 - val_loss: 1.3269 - val_accuracy: 0.7020
Epoch 3/9
16/16 [==============================] - 1s 39ms/step - loss: 1.1155 - accuracy: 0.7
615 - val_loss: 1.1361 - val_accuracy: 0.7490
Epoch 4/9
16/16 [==============================] - 1s 38ms/step - loss: 0.8418 - accuracy: 0.8
249 - val_loss: 1.0924 - val_accuracy: 0.7470
Epoch 5/9
16/16 [==============================] - 1s 41ms/step - loss: 0.6969 - accuracy: 0.8
511 - val_loss: 0.9652 - val_accuracy: 0.7970
Epoch 6/9
16/16 [==============================] - 1s 36ms/step - loss: 0.5225 - accuracy: 0.8
975 - val_loss: 0.9553 - val_accuracy: 0.8090
Epoch 7/9
16/16 [==============================] - 1s 37ms/step - loss: 0.4331 - accuracy: 0.9
154 - val_loss: 0.9011 - val_accuracy: 0.8070
Epoch 8/9
16/16 [==============================] - 1s 38ms/step - loss: 0.3354 - accuracy: 0.9
309 - val_loss: 0.8921 - val_accuracy: 0.8250
Epoch 9/9
16/16 [==============================] - 1s 37ms/step - loss: 0.2770 - accuracy: 0.9
400 - val_loss: 0.8988 - val_accuracy: 0.8220


---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call last)
<ipython-input-24-79e5579873cc> in <module>
     14          validation_data=(X_val, y_val))
     15
---> 16 results = model.evaluate(x_test, one_hot_test_labels)

NameError: name 'x_test' is not defined
```

In [ ]:

```
results
```

In [25]:

```
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[25]:

0.19011576135351738

새로운 데이터로 예측

In [27]:

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

In [28]:

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

Out[28]:

(46,)

In [29]:

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

Out[29]:

1.0000001

가장 큰 값이 예측 클래스가 된다.

In [30]:

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

Out[30]:

3

정수 레이블(타켓)을 그대로 사용할 때

- 손실함수를 sparse_categorical_crossentropy 를 사용

In [32]:

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

print(y_train.shape)
```

```
(8982,)
```

In [33]:

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

In [35]:

```
X_val = X_train[ :1000]
partial_x_train = X_train[1000:]

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

In [36]:

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

Out[36]:

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

In [38]:

```
## 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 [==============================] - 2s 59ms/step - loss: 0.2749 - acc: 0.9341 -
val_loss: 0.9116 - val_acc: 0.8200
Epoch 2/20
16/16 [==============================] - 1s 38ms/step - loss: 0.1981 - acc: 0.9541 -
val_loss: 0.9963 - val_acc: 0.8110
Epoch 3/20
16/16 [==============================] - 1s 40ms/step - loss: 0.1812 - acc: 0.9541 -
val_loss: 0.9674 - val_acc: 0.8110
Epoch 4/20
16/16 [==============================] - 1s 38ms/step - loss: 0.1578 - acc: 0.9569 -
val_loss: 0.9871 - val_acc: 0.8070
Epoch 5/20
16/16 [==============================] - 1s 37ms/step - loss: 0.1460 - acc: 0.9576 -
val_loss: 1.0395 - val_acc: 0.8010
Epoch 6/20
16/16 [==============================] - 1s 36ms/step - loss: 0.1254 - acc: 0.9578 -
val_loss: 1.0376 - val_acc: 0.8000
Epoch 7/20
16/16 [==============================] - 1s 38ms/step - loss: 0.1232 - acc: 0.9608 -
val_loss: 1.0883 - val_acc: 0.7860
Epoch 8/20
16/16 [==============================] - 1s 39ms/step - loss: 0.1185 - acc: 0.9602 -
val_loss: 1.0381 - val_acc: 0.8090
Epoch 9/20
16/16 [==============================] - 1s 37ms/step - loss: 0.1084 - acc: 0.9585 -
val_loss: 1.0608 - val_acc: 0.8070
Epoch 10/20
16/16 [==============================] - 1s 39ms/step - loss: 0.1074 - acc: 0.9608 -
val_loss: 1.1047 - val_acc: 0.7950
Epoch 11/20
16/16 [==============================] - 1s 36ms/step - loss: 0.0964 - acc: 0.9626 -
val_loss: 1.1285 - val_acc: 0.7930
Epoch 12/20
16/16 [==============================] - 1s 39ms/step - loss: 0.1049 - acc: 0.9599 -
val_loss: 1.1653 - val_acc: 0.7950
Epoch 13/20
16/16 [==============================] - 1s 36ms/step - loss: 0.0935 - acc: 0.9616 -
val_loss: 1.1215 - val_acc: 0.7970
Epoch 14/20
16/16 [==============================] - 1s 37ms/step - loss: 0.0868 - acc: 0.9657 -
val_loss: 1.1203 - val_acc: 0.8030
Epoch 15/20
16/16 [==============================] - 1s 38ms/step - loss: 0.0931 - acc: 0.9610 -
val_loss: 1.1632 - val_acc: 0.7900
Epoch 16/20
16/16 [==============================] - 1s 38ms/step - loss: 0.0930 - acc: 0.9635 -
val_loss: 1.2006 - val_acc: 0.7960
Epoch 17/20
16/16 [==============================] - 1s 38ms/step - loss: 0.0936 - acc: 0.9591 -
val_loss: 1.1871 - val_acc: 0.7980
Epoch 18/20
```

```
16/16 [==============================] - 1s 39ms/step - loss: 0.0829 - acc: 0.9636 -
val_loss: 1.1703 - val_acc: 0.8020
Epoch 19/20
16/16 [==============================] - 1s 38ms/step - loss: 0.0846 - acc: 0.9627 -
val_loss: 1.2287 - val_acc: 0.7890
Epoch 20/20
16/16 [==============================] - 1s 36ms/step - loss: 0.0827 - acc: 0.9646 -
val_loss: 1.2739 - val_acc: 0.7880
```

충분히 큰 중간층을 두기

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

In [39]:

```
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 [40]:

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

In [41]:

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

Out[41]:

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

In [42]:

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

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-42-a7052cd7a810> in <module>
      2                     epochs=20,
      3                     batch_size=512,
----> 4                     validation_data=(x_val, y_val))

NameError: name 'x_val' is not defined
```

46차원보다 훨씬 작은 중간층(예를 들어 4차원)을 두면,

In [ ]:

```
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))
```

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

In [ ]: