뉴스 기사 분류: 다중 분류 문제 ¶

학습 내용

- 01

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)
```

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

In [3]:

```
len(train_data)
```

Out[3]:

8982

In [4]:

```
len(test_data)
```

Out[4]:

2246

In [5]:

```
train_data[10]
```

Out[5]:

```
[1,
 245,
 273,
 207,
 156,
 53,
 74,
 160,
 26,
 14,
 46,
 296,
 26,
 39,
 74,
 2979,
 3554,
 14,
 46,
 4689,
 4329,
 86,
 61,
 3499,
 4795,
 14,
 61,
 451,
 4329,
 17,
 12]
```

단어로 디코딩

In [6]:

```
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 [7]:

```
decoded_newswire
```

Out[7]:

```
'? ? ? 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 [8]:

```
train_labels[10]
```

Out[8]:

```
3
```

데이터 준비

In [9]:

```
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)
```

레이블을 벡터로 바꾸는 방법은 두 가지

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

In [10]:

```
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

# 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 Dense 층의 크기가 46 : 각 입력 샘플에 대해서 46차원의 벡터를 출력
- 마지막 층에 softmax 활성화 함수가 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 [11]:

```
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 [12]:

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

모델 검증

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

In [13]:

```
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 [14]:

```
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 [==============================] - 1s 69ms/step - loss: 2.6841 - accuracy: 0.4
783 - val_loss: 1.7817 - val_accuracy: 0.6300
Epoch 2/20
16/16 [==============================] - 1s 33ms/step - loss: 1.4661 - accuracy: 0.6
963 - val_loss: 1.3314 - val_accuracy: 0.7070
Epoch 3/20
16/16 [==============================] - 1s 36ms/step - loss: 1.0796 - accuracy: 0.7
715 - val_loss: 1.1496 - val_accuracy: 0.7520
Epoch 4/20
16/16 [==============================] - 1s 36ms/step - loss: 0.8481 - accuracy: 0.8
257 - val_loss: 1.0387 - val_accuracy: 0.7860
Epoch 5/20
16/16 [==============================] - 1s 36ms/step - loss: 0.6775 - accuracy: 0.8
614 - val_loss: 0.9844 - val_accuracy: 0.7960
Epoch 6/20
16/16 [==============================] - 1s 33ms/step - loss: 0.5387 - accuracy: 0.8
914 - val_loss: 0.9430 - val_accuracy: 0.7960
Epoch 7/20
16/16 [==============================] - 1s 35ms/step - loss: 0.4322 - accuracy: 0.9
107 - val_loss: 0.9220 - val_accuracy: 0.8070
Epoch 8/20
16/16 [==============================] - 1s 35ms/step - loss: 0.3515 - accuracy: 0.9
270 - val_loss: 0.9179 - val_accuracy: 0.8110
Epoch 9/20
16/16 [==============================] - 1s 32ms/step - loss: 0.2907 - accuracy: 0.9
366 - val_loss: 0.9485 - val_accuracy: 0.8030
Epoch 10/20
16/16 [==============================] - 1s 32ms/step - loss: 0.2448 - accuracy: 0.9
439 - val_loss: 0.9354 - val_accuracy: 0.8080
Epoch 11/20
16/16 [==============================] - 1s 33ms/step - loss: 0.2148 - accuracy: 0.9
479 - val_loss: 0.9444 - val_accuracy: 0.8090
Epoch 12/20
16/16 [==============================] - 1s 33ms/step - loss: 0.1878 - accuracy: 0.9
493 - val_loss: 0.9370 - val_accuracy: 0.8090
Epoch 13/20
16/16 [==============================] - 1s 33ms/step - loss: 0.1715 - accuracy: 0.9
520 - val_loss: 0.9690 - val_accuracy: 0.8100
Epoch 14/20
16/16 [==============================] - 1s 33ms/step - loss: 0.1551 - accuracy: 0.9
534 - val_loss: 1.0032 - val_accuracy: 0.8020
Epoch 15/20
16/16 [==============================] - 1s 33ms/step - loss: 0.1430 - accuracy: 0.9
557 - val_loss: 0.9768 - val_accuracy: 0.8150
Epoch 16/20
16/16 [==============================] - 1s 32ms/step - loss: 0.1327 - accuracy: 0.9
574 - val_loss: 1.0386 - val_accuracy: 0.8090
Epoch 17/20
16/16 [==============================] - 1s 32ms/step - loss: 0.1273 - accuracy: 0.9
565 - val_loss: 1.0643 - val_accuracy: 0.8070
Epoch 18/20
```

```
16/16 [==============================] - 1s 32ms/step - loss: 0.1210 - accuracy: 0.9
562 - val_loss: 1.1337 - val_accuracy: 0.7960
Epoch 19/20
16/16 [==============================] - 0s 31ms/step - loss: 0.1154 - accuracy: 0.9
572 - val_loss: 1.1028 - val_accuracy: 0.8040
Epoch 20/20
16/16 [==============================] - 0s 31ms/step - loss: 0.1153 - accuracy: 0.9
583 - val_loss: 1.1705 - val_accuracy: 0.7880
```

손실과 정확도 곡선

In [15]:

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

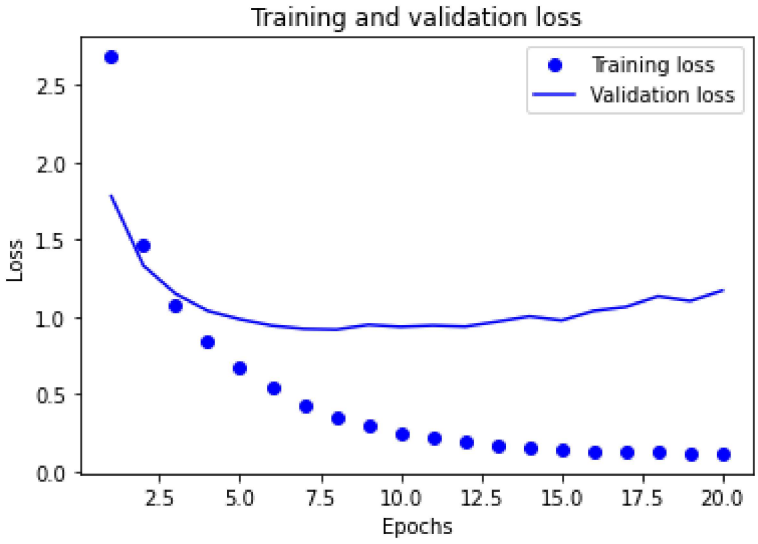
In [16]:

```
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()
```



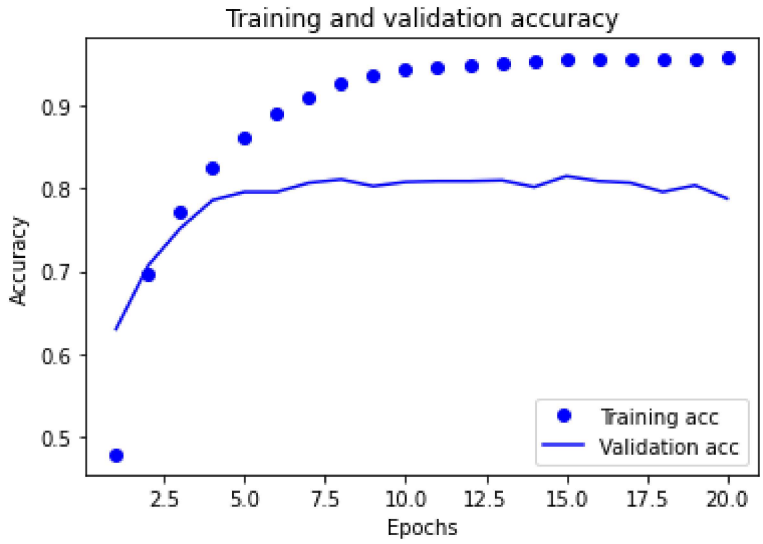
In [18]:

```
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 [19]:

```
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 [==============================] - 1s 45ms/step - loss: 2.4499 - accuracy: 0.5
273 - val_loss: 1.6193 - val_accuracy: 0.6600
Epoch 2/9
16/16 [==============================] - 1s 32ms/step - loss: 1.3470 - accuracy: 0.6
996 - val_loss: 1.2831 - val_accuracy: 0.7000
Epoch 3/9
16/16 [==============================] - 1s 34ms/step - loss: 1.0199 - accuracy: 0.7
759 - val_loss: 1.1345 - val_accuracy: 0.7450
Epoch 4/9
16/16 [==============================] - 1s 43ms/step - loss: 0.8075 - accuracy: 0.8
261 - val_loss: 1.0245 - val_accuracy: 0.7820
Epoch 5/9
16/16 [==============================] - 1s 49ms/step - loss: 0.6443 - accuracy: 0.8
654 - val_loss: 0.9820 - val_accuracy: 0.7950
Epoch 6/9
16/16 [==============================] - 1s 49ms/step - loss: 0.5149 - accuracy: 0.8
946 - val_loss: 0.9152 - val_accuracy: 0.8090
Epoch 7/9
16/16 [==============================] - 1s 43ms/step - loss: 0.4187 - accuracy: 0.9
116 - val_loss: 0.8849 - val_accuracy: 0.8120
Epoch 8/9
16/16 [==============================] - 1s 44ms/step - loss: 0.3393 - accuracy: 0.9
283 - val_loss: 0.8990 - val_accuracy: 0.8070
Epoch 9/9
16/16 [==============================] - 1s 41ms/step - loss: 0.2829 - accuracy: 0.9
392 - val_loss: 0.9363 - val_accuracy: 0.8030
71/71 [==============================] - 0s 3ms/step - loss: 1.0100 - accuracy: 0.78
09
```

In [20]:

```
results
```

Out[20]:

```
[1.0099729299545288, 0.7809438705444336]
```

In [21]:

```
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[21]:

0.1861086375779163

새로운 데이터로 예측

In [22]:

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

In [23]:

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

Out[23]:

(46,)

In [24]:

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

Out[24]:

0.9999999

가장 큰 값이 예측 클래스가 된다.

In [27]:

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

Out[27]:

3

정수 레이블을 사용할 때

In [29]:

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

print(y_train.shape)
```

```
(8982,)
```

In [28]:

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

충분히 큰 중간층을 두기

In [30]:

```
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='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(partial_x_train,
          partial_y_train,
          epochs=20,
          batch_size=128,
          validation_data=(x_val, y_val))
```

```
Epoch 1/20
63/63 [==============================] - 1s 23ms/step - loss: 2.6927 - accuracy: 0.3
413 - val_loss: 2.0556 - val_accuracy: 0.4240
Epoch 2/20
63/63 [==============================] - 1s 16ms/step - loss: 1.8412 - accuracy: 0.5
510 - val_loss: 1.7208 - val_accuracy: 0.5740
Epoch 3/20
63/63 [==============================] - 1s 16ms/step - loss: 1.5611 - accuracy: 0.5
921 - val_loss: 1.5860 - val_accuracy: 0.6080
Epoch 4/20
63/63 [==============================] - 1s 17ms/step - loss: 1.3840 - accuracy: 0.6
319 - val_loss: 1.4995 - val_accuracy: 0.6270
Epoch 5/20
63/63 [==============================] - 1s 17ms/step - loss: 1.2539 - accuracy: 0.6
798 - val_loss: 1.4678 - val_accuracy: 0.6450
Epoch 6/20
63/63 [==============================] - 1s 16ms/step - loss: 1.1576 - accuracy: 0.6
991 - val_loss: 1.4376 - val_accuracy: 0.6620
Epoch 7/20
63/63 [==============================] - 1s 16ms/step - loss: 1.0796 - accuracy: 0.7
091 - val_loss: 1.4423 - val_accuracy: 0.6610
Epoch 8/20
63/63 [==============================] - 1s 15ms/step - loss: 1.0185 - accuracy: 0.7
273 - val_loss: 1.4789 - val_accuracy: 0.6730
Epoch 9/20
63/63 [==============================] - 1s 15ms/step - loss: 0.9685 - accuracy: 0.7
459 - val_loss: 1.4651 - val_accuracy: 0.6820
Epoch 10/20
63/63 [==============================] - 1s 16ms/step - loss: 0.9240 - accuracy: 0.7
552 - val_loss: 1.5005 - val_accuracy: 0.6830
Epoch 11/20
63/63 [==============================] - 1s 16ms/step - loss: 0.8864 - accuracy: 0.7
612 - val_loss: 1.5538 - val_accuracy: 0.6810
Epoch 12/20
63/63 [==============================] - 1s 16ms/step - loss: 0.8507 - accuracy: 0.7
689 - val_loss: 1.5704 - val_accuracy: 0.6790
Epoch 13/20
63/63 [==============================] - 1s 16ms/step - loss: 0.8219 - accuracy: 0.7
702 - val_loss: 1.6332 - val_accuracy: 0.6800
Epoch 14/20
63/63 [==============================] - 1s 18ms/step - loss: 0.7940 - accuracy: 0.7
742 - val_loss: 1.6701 - val_accuracy: 0.6730
Epoch 15/20
63/63 [==============================] - 1s 18ms/step - loss: 0.7699 - accuracy: 0.7
```

```
806 - val_loss: 1.7212 - val_accuracy: 0.6780
Epoch 16/20
63/63 [==============================] - 1s 18ms/step - loss: 0.7499 - accuracy: 0.7
834 - val_loss: 1.7664 - val_accuracy: 0.6750
Epoch 17/20
63/63 [==============================] - 1s 17ms/step - loss: 0.7309 - accuracy: 0.7
866 - val_loss: 1.8025 - val_accuracy: 0.6700
Epoch 18/20
63/63 [==============================] - 1s 18ms/step - loss: 0.7152 - accuracy: 0.7
905 - val_loss: 1.8662 - val_accuracy: 0.6770
Epoch 19/20
63/63 [==============================] - 1s 18ms/step - loss: 0.6975 - accuracy: 0.7
917 - val_loss: 1.8537 - val_accuracy: 0.6780
Epoch 20/20
63/63 [==============================] - 1s 18ms/step - loss: 0.6848 - accuracy: 0.8
016 - val_loss: 2.0017 - val_accuracy: 0.6760
```

Out[30]:

```
<tensorflow.python.keras.callbacks.History at 0x2598cf5b070>
```

검증 정확도가 감소

- 검증 정확도의 최고 값은 약 67%로 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 [ ]: