뉴스 기사 분류: 다중 분류 문제

학습 내용

- 01 로이터 뉴스를 이용한 신경망 구현
- 02 다항분류 문제
  - 46개의 클래스로 구분(46개의 Topic를 예측하는 딥러닝 모델)

목차

In [1]:

```
import keras
import numpy as np
import matplotlib.pyplot as plt

keras.__version__
```

Out[1]:

```
'2.9.0'
```

01 데이터 설명 및 개요

로이터 뉴스를 46개의 상호 배타적인 토픽으로 분류하는 신경망

- 1986년에 로이터에서 공개한 짧은 뉴스 기사와 토픽의 집합인 로이터 데이터셋을 사용
- 46개의 토픽
- 각 토픽은 학습용 세트에 최소한 10개의 샘플
- 원본 Reuters Dataset : https://martin-thoma.com/nlp-reuters/ (https://martin-thoma.com/nlp-reuters/)
  - 90 classes, 7769 training document, 3019 test documents

뉴스의 46개의 토픽 주제

['cocoa','grain','veg-oil','earn','acq','wheat','copper','housing','money-supply',
'coffee','sugar','trade','reserves','ship','cotton','carcass','crude','nat-gas', 'cpi','money-fx','interest','gnp','meal-feed','alum','oilseed','gold','tin', 'strategic-metal','livestock','retail','ipi','iron-steel','rubber','heat','jobs',
'lei','bop','zinc','orange','pet-chem','dlr','gas','silver','wpi','hog','lead']

데이터 가져오기

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]:

```
# 데이터 뉴스의 개수
len(train_data), len(test_data), len(train_data)+ len(test_data)
```

Out[3]:

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

In [4]:

```
# 46개의 토픽
print( np.unique(train_labels) )
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45]
```

In [5]:

```
# 2번째 뉴스 데이터 확인
print("데이터의 길이 : ", len(train_data[1]) )

# 처음부터 14개의 단어 인덱스 확인
print("데이터 내용(숫자) :", train_data[1][0:15] )
```

```
데이터의 길이 :  56
데이터 내용(숫자) : [1, 3267, 699, 3434, 2295, 56, 2, 7511, 9, 56, 3906, 1073, 81,
5, 1198]
```

02 실제 뉴스 데이터 내용 확인

숫자 데이터를 단어로 확인해 보자.

In [6]:

```
print( "자료형 : ", type(reuters) )
print( "reuters의 기능 리스트 : ", dir(reuters) )
```

```
자료형 :  <class 'module'>
reuters의 기능 리스트 :  ['__builtins__', '__cached__', '__doc__', '__file__', '__lo
ader__', '__name__', '__package__', '__spec__', '_remove_long_seq', 'get_file', 'get
_word_index', 'json', 'keras_export', 'load_data', 'logging', 'np']
```

reuters의 함수 get_word_index() 를 이용하여 정보 획득

- 단어:인덱스번호 형태로 이루어져 있음.

In [7]:

```
# reuters의 word의 index를 얻기
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 [8]:

```
# 첫번째 뉴스 기사(숫자로 되어 있음)를 영문 매칭된 단어로 변경
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)
```

03 모델 사용을 위한 데이터 전처리

- 입력 데이터 전처리
  - 각각의 뉴스는 리스트로 이루어져 있음. 이를 딥러닝 모델에 맞추어 10000개의 차원으로 이루어진 벡터로 변환
  - 0D 텐서 -> 1D 텐서(벡터) 형태 변환

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)
```

출력(레이블) 데이터 전처리 2가지 방법

- 01 원-핫 인코딩을 사용하여 스칼라 -> 1D텐서(벡터)로 변환
- 02 레이블의 토픽값(정수)를 그대로 이용

01 원핫 인코딩으로 46개 차원의 벡터형태로 변환

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
y_train = to_one_hot(train_labels)

# 테스트 레이블 벡터 변환
y_test = to_one_hot(test_labels)

print("변환 전 : ", train_labels.shape, test_labels.shape)
print("변환 후 : ", y_train.shape, y_test.shape)
```

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

- MNIST 예제에서 이미 보았듯이 케라스에는 이를 위한 내장 함수

04 모델 구축 및 학습, 평가

- 마지막 출력이 46차원이기 때문에 중간층의 히든 유닛이 46개보다 많이 적어서는 안된다.
  - 은닉층의 수가 많이 적을 때, 성능의 차이가 있을 수 있다.
- 마지막 Dense 층의 크기가 46 : 각 입력 샘플에 대해서 46차원의 벡터를 출력
- 마지막 층은 다항 분류의 경우, 활성화 함수로 **softmax 활성화 함수**을 사용
- 각 입력 샘플마다 **46개의 출력 클래스에 대한 확률 분포**를 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

실습해 보기

- (1) 가장 간단한 모델 구성
- (2) 모델 학습을 위한 학습, 검증용 데이터 셋을 나누어 학습 시켜보기

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'))
```

모델 세부 설정

- 최적화 함수(optimizer) : rmsprop
- 오차 함수(loss) : categorical_crossentropy
- 평가 지표(metrics) : 정확도(accuracy)

In [13]:

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

모델 검증

- 학습용 데이터 8,892개에서 1,000개의 샘플을 따로 떼어서 검증 세트로 사용
- 학습(X), 테스트(X), 학습(y), 테스트(y)의 데이터 나누기를 다음과 같이 좀 더 세분화
  - 학습(X) => 자체학습(X_train), 자체검증(X_val)
  - 학습(y) => 자체학습(y_train), 자체검증(y_val)
  - 테스트(X), 테스트(y)는 딥러닝 모델 학습 완료 후, 최종적으로 확인을 위해 사용.

In [14]:

```
X_val = X_train[:1000]    # 0~8982 -> 0~1000개를 검증
partial_X_train = X_train[1000:] # 0~8982 -> 1000~끝까지 학습

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

모델 학습

In [15]:

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

In [16]:

```
# early_stopping 를 이용하여 성능의 개선이 없을 시, 모델 학습을 중지
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 [==============================] - 1s 35ms/step - loss: 2.6155 - accuracy: 0.5
471 - val_loss: 1.6510 - val_accuracy: 0.6560
Epoch 2/50
16/16 [==============================] - 0s 22ms/step - loss: 1.3679 - accuracy: 0.7
144 - val_loss: 1.2934 - val_accuracy: 0.7120
Epoch 3/50
16/16 [==============================] - 0s 22ms/step - loss: 1.0260 - accuracy: 0.7
759 - val_loss: 1.1505 - val_accuracy: 0.7370
Epoch 4/50
16/16 [==============================] - 0s 22ms/step - loss: 0.8151 - accuracy: 0.8
234 - val_loss: 1.0369 - val_accuracy: 0.7790
Epoch 5/50
16/16 [==============================] - 0s 22ms/step - loss: 0.6469 - accuracy: 0.8
576 - val_loss: 0.9757 - val_accuracy: 0.7910
Epoch 6/50
16/16 [==============================] - 0s 22ms/step - loss: 0.5190 - accuracy: 0.8
879 - val_loss: 0.9298 - val_accuracy: 0.8070
Epoch 7/50
16/16 [==============================] - 0s 22ms/step - loss: 0.4175 - accuracy: 0.9
108 - val_loss: 0.9005 - val_accuracy: 0.8110
Epoch 8/50
16/16 [==============================] - 0s 22ms/step - loss: 0.3328 - accuracy: 0.9
277 - val_loss: 0.8854 - val_accuracy: 0.8210
Epoch 9/50
16/16 [==============================] - 0s 22ms/step - loss: 0.2837 - accuracy: 0.9
357 - val_loss: 0.8901 - val_accuracy: 0.8220
Epoch 10/50
16/16 [==============================] - 0s 21ms/step - loss: 0.2357 - accuracy: 0.9
436 - val_loss: 0.9105 - val_accuracy: 0.8130
Epoch 11/50
16/16 [==============================] - 0s 22ms/step - loss: 0.2049 - accuracy: 0.9
499 - val_loss: 0.9153 - val_accuracy: 0.8090
Epoch 12/50
16/16 [==============================] - 0s 22ms/step - loss: 0.1820 - accuracy: 0.9
515 - val_loss: 0.9494 - val_accuracy: 0.8120
Epoch 13/50
16/16 [==============================] - 0s 23ms/step - loss: 0.1624 - accuracy: 0.9
519 - val_loss: 0.9889 - val_accuracy: 0.8010
Epoch 14/50
16/16 [==============================] - 0s 24ms/step - loss: 0.1472 - accuracy: 0.9
543 - val_loss: 0.9829 - val_accuracy: 0.8100
Epoch 15/50
16/16 [==============================] - 0s 23ms/step - loss: 0.1423 - accuracy: 0.9
573 - val_loss: 1.0094 - val_accuracy: 0.8010
Epoch 16/50
16/16 [==============================] - 0s 24ms/step - loss: 0.1293 - accuracy: 0.9
572 - val_loss: 0.9937 - val_accuracy: 0.8120
Epoch 17/50
16/16 [==============================] - 0s 23ms/step - loss: 0.1259 - accuracy: 0.9
555 - val_loss: 1.0202 - val_accuracy: 0.8090
```

```
Epoch 18/50
16/16 [==============================] - 0s 23ms/step - loss: 0.1181 - accuracy: 0.9
574 - val_loss: 1.0980 - val_accuracy: 0.7980
Epoch 19/50
16/16 [==============================] - 0s 23ms/step - loss: 0.1157 - accuracy: 0.9
541 - val_loss: 1.0604 - val_accuracy: 0.8110
Epoch 20/50
16/16 [==============================] - 0s 22ms/step - loss: 0.1139 - accuracy: 0.9
559 - val_loss: 1.0643 - val_accuracy: 0.8040
Epoch 21/50
16/16 [==============================] - 0s 23ms/step - loss: 0.1094 - accuracy: 0.9
583 - val_loss: 1.1420 - val_accuracy: 0.8010
Epoch 22/50
16/16 [==============================] - 0s 21ms/step - loss: 0.1092 - accuracy: 0.9
577 - val_loss: 1.0772 - val_accuracy: 0.8070
Epoch 23/50
16/16 [==============================] - 0s 22ms/step - loss: 0.1038 - accuracy: 0.9
597 - val_loss: 1.1157 - val_accuracy: 0.8050
Epoch 24/50
16/16 [==============================] - 0s 22ms/step - loss: 0.1018 - accuracy: 0.9
577 - val_loss: 1.1744 - val_accuracy: 0.7890
Epoch 25/50
16/16 [==============================] - 0s 22ms/step - loss: 0.0992 - accuracy: 0.9
587 - val_loss: 1.1362 - val_accuracy: 0.8010
Epoch 26/50
16/16 [==============================] - 0s 22ms/step - loss: 0.1005 - accuracy: 0.9
595 - val_loss: 1.1583 - val_accuracy: 0.7980
Epoch 27/50
16/16 [==============================] - 0s 22ms/step - loss: 0.0965 - accuracy: 0.9
585 - val_loss: 1.1892 - val_accuracy: 0.7940
Epoch 28/50
16/16 [==============================] - 0s 23ms/step - loss: 0.0952 - accuracy: 0.9
587 - val_loss: 1.1497 - val_accuracy: 0.8030
```

In [17]:

```
results = model.evaluate(X_test, y_test)
print("최종 평가(loss, accuracy) : ", results)
```

```
71/71 [==============================] - 0s 2ms/step - loss: 1.3768 - accuracy: 0.78
50
최종 평가(loss, accuracy) :  [1.376828670501709, 0.7849510312080383]
```

05 평가 결과 시각화

In [18]:

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

에폭별 loss 결과 시각화

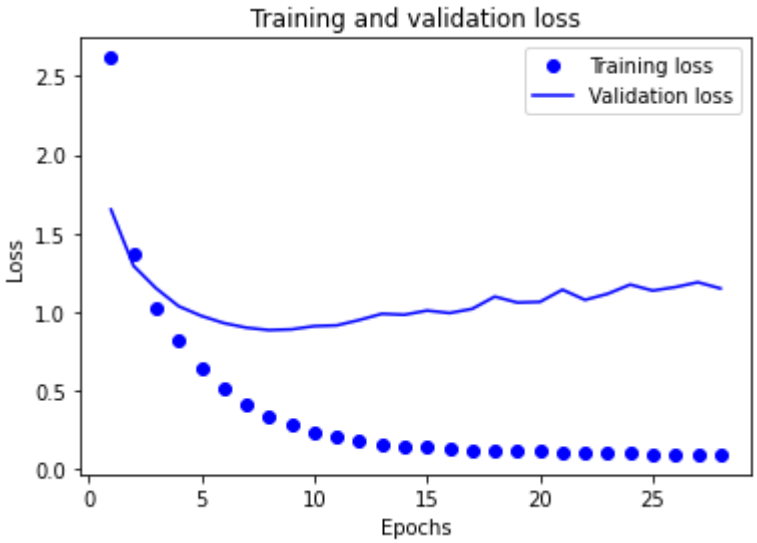
In [19]:

```
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()
```



에폭별 accuracy(정확도) 결과 시각화

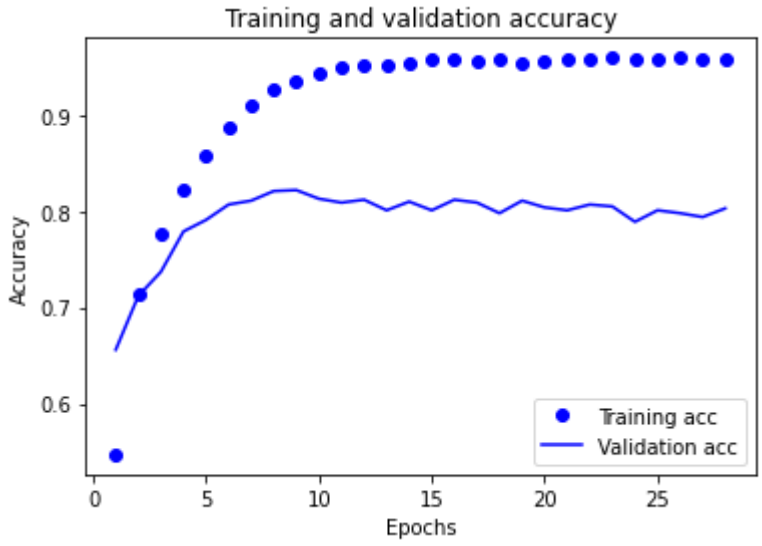
In [20]:

```
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()
```



06 적당한 epoch으로 조정

모델 변경 - epochs을 변경

- 9번째 에포크 이후에 과대적합이 현상이 보임.
- 9번의 에포크로 새로운 모델 훈련과 테스트 세트에서 평가

In [21]:

```
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))
```

```
Epoch 1/9
16/16 [==============================] - 1s 30ms/step - loss: 2.4414 - accuracy: 0.5
509 - val_loss: 1.6142 - val_accuracy: 0.6460
Epoch 2/9
16/16 [==============================] - 0s 22ms/step - loss: 1.3279 - accuracy: 0.7
201 - val_loss: 1.2674 - val_accuracy: 0.7240
Epoch 3/9
16/16 [==============================] - 0s 21ms/step - loss: 1.0040 - accuracy: 0.7
851 - val_loss: 1.1093 - val_accuracy: 0.7580
Epoch 4/9
16/16 [==============================] - 0s 22ms/step - loss: 0.7914 - accuracy: 0.8
276 - val_loss: 1.0082 - val_accuracy: 0.7820
Epoch 5/9
16/16 [==============================] - 0s 22ms/step - loss: 0.6317 - accuracy: 0.8
611 - val_loss: 0.9448 - val_accuracy: 0.8110
Epoch 6/9
16/16 [==============================] - 0s 22ms/step - loss: 0.5058 - accuracy: 0.8
931 - val_loss: 0.9014 - val_accuracy: 0.8090
Epoch 7/9
16/16 [==============================] - 0s 23ms/step - loss: 0.4059 - accuracy: 0.9
129 - val_loss: 0.8961 - val_accuracy: 0.8100
Epoch 8/9
16/16 [==============================] - 0s 23ms/step - loss: 0.3313 - accuracy: 0.9
280 - val_loss: 0.9186 - val_accuracy: 0.8150
Epoch 9/9
16/16 [==============================] - 0s 21ms/step - loss: 0.2742 - accuracy: 0.9
379 - val_loss: 0.8857 - val_accuracy: 0.8200
```

Out[21]:

```
<keras.callbacks.History at 0x1f39049f1f0>
```

In [22]:

```
results = model.evaluate(X_test, y_test)
print("최종 평가(loss, accuracy) : ", results)
```

```
71/71 [==============================] - 0s 2ms/step - loss: 0.9602 - accuracy: 0.79
39
최종 평가(loss, accuracy) :  [0.9601863026618958, 0.7938557267189026]
```

모델 학습 후, 테스트 데이터를 사용하여 예측

In [25]:

```
pred = model.predict(X_test)
pred[0]
```

```
71/71 [==============================] - 0s 2ms/step
```

Out[25]:

```
array([1.61659154e-05, 1.00647594e-04, 8.32144542e-06, 8.74383152e-01,
       9.08382311e-02, 1.49808810e-04, 7.12968700e-04, 3.55729717e-05,
       4.15602187e-03, 1.17137288e-05, 5.94606827e-05, 6.56019431e-04,
       2.45071802e-04, 1.55638409e-04, 9.83425416e-05, 7.67887614e-05,
       1.96631998e-03, 5.41928515e-04, 3.09917988e-04, 3.92658776e-03,
       1.53046707e-02, 5.63942769e-04, 1.01047968e-04, 4.01969271e-04,
       1.88929534e-05, 4.22453231e-05, 1.65694000e-05, 1.66216356e-04,
       8.48767580e-04, 8.33748782e-04, 1.22651923e-04, 2.72798032e-04,
       7.34142668e-05, 5.85259613e-06, 6.27420843e-04, 4.90616840e-05,
       3.93881433e-04, 4.44498291e-04, 6.11636506e-06, 9.19212936e-04,
       1.73625776e-05, 2.65926006e-04, 2.08015208e-05, 1.64218091e-05,
       1.46543789e-06, 1.61237604e-05], dtype=float32)
```

In [26]:

```
# 첫번째 데이터의 확률 분포 확인
pred[0].shape
```

Out[26]:

```
(46,)
```

In [28]:

```
# softmax 활성화 함수 사용 - 46개 뉴런의 예측 확률의 합은 1이 된다(softmax)
np.sum(pred[0])
```

Out[28]:

```
0.9999998
```

확률이 가장 큰 값이 예측 클래스가 된다.

In [29]:

```
# 확률이 가장 큰 값을 예측 클래스로 한다.
np.argmax(pred[0])
```

Out[29]:

```
3
```

06 다양한 모델(오차함수 변경)

- 정수 레이블(타켓)을 그대로 사용할 때
  - 손실함수를 sparse_categorical_crossentropy 를 사용

정수 레이블을 그대로 이용

In [30]:

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

print(y_train.shape)

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

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

(8982,)

In [31]:

```
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='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(partial_X_train,
          partial_y_train,
          epochs=9,
          batch_size=512,
          validation_data=(X_val, y_val))
```

```
Epoch 1/9
16/16 [==============================] - 1s 30ms/step - loss: 2.6721 - accuracy: 0.5
356 - val_loss: 1.7506 - val_accuracy: 0.6370
Epoch 2/9
16/16 [==============================] - 0s 22ms/step - loss: 1.4352 - accuracy: 0.7
110 - val_loss: 1.3116 - val_accuracy: 0.7220
Epoch 3/9
16/16 [==============================] - 0s 22ms/step - loss: 1.0633 - accuracy: 0.7
742 - val_loss: 1.1400 - val_accuracy: 0.7430
Epoch 4/9
16/16 [==============================] - 0s 22ms/step - loss: 0.8358 - accuracy: 0.8
176 - val_loss: 1.0478 - val_accuracy: 0.7710
Epoch 5/9
16/16 [==============================] - 0s 22ms/step - loss: 0.6672 - accuracy: 0.8
597 - val_loss: 0.9760 - val_accuracy: 0.8030
Epoch 6/9
16/16 [==============================] - 0s 22ms/step - loss: 0.5312 - accuracy: 0.8
875 - val_loss: 0.9365 - val_accuracy: 0.8080
Epoch 7/9
16/16 [==============================] - 0s 22ms/step - loss: 0.4291 - accuracy: 0.9
098 - val_loss: 0.9055 - val_accuracy: 0.8160
Epoch 8/9
16/16 [==============================] - 0s 23ms/step - loss: 0.3446 - accuracy: 0.9
276 - val_loss: 0.8911 - val_accuracy: 0.8250
Epoch 9/9
16/16 [==============================] - 0s 20ms/step - loss: 0.2857 - accuracy: 0.9
365 - val_loss: 0.9134 - val_accuracy: 0.8210
```

Out[31]:

```
<keras.callbacks.History at 0x1f391881700>
```

In [32]:

```
results = model.evaluate(X_test, y_test)
print("최종 평가(loss, accuracy) : ", results)
```

```
71/71 [==============================] - 0s 2ms/step - loss: 1.0036 - accuracy: 0.79
16
최종 평가(loss, accuracy) :  [1.003587007522583, 0.7916295528411865]
```

08 다양한 모델(은닉층 뉴런이 많이 작을 경우)

충분히 큰 중간층을 두기

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

In [33]:

```
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 [34]:

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

print( partial_x_train.shape, partial_y_train.shape )

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

results = model.evaluate(X_test, y_test)
print("최종 평가(loss, accuracy) : ", results)
```

```
(7982, 10000) (7982,)
Epoch 1/20
16/16 [==============================] - 1s 30ms/step - loss: 2.6433 - accuracy:
0.5163 - val_loss: 1.7144 - val_accuracy: 0.6160
Epoch 2/20
16/16 [==============================] - 0s 22ms/step - loss: 1.4106 - accuracy:
0.7057 - val_loss: 1.3014 - val_accuracy: 0.7050
Epoch 3/20
16/16 [==============================] - 0s 22ms/step - loss: 1.0596 - accuracy:
0.7751 - val_loss: 1.1309 - val_accuracy: 0.7540
Epoch 4/20
16/16 [==============================] - 0s 22ms/step - loss: 0.8405 - accuracy:
0.8225 - val_loss: 1.0333 - val_accuracy: 0.7880
Epoch 5/20
16/16 [==============================] - 0s 22ms/step - loss: 0.6775 - accuracy:
0.8578 - val_loss: 0.9855 - val_accuracy: 0.8080
Epoch 6/20
16/16 [==============================] - 0s 22ms/step - loss: 0.5481 - accuracy:
0.8881 - val_loss: 0.9285 - val_accuracy: 0.8090
Epoch 7/20
16/16 [==============================] - 0s 21ms/step - loss: 0.4422 - accuracy:
0.9060 - val_loss: 0.9059 - val_accuracy: 0.8070
Epoch 8/20
16/16 [==============================] - 0s 23ms/step - loss: 0.3642 - accuracy:
0.9227 - val_loss: 0.9373 - val_accuracy: 0.8030
Epoch 9/20
16/16 [==============================] - 0s 22ms/step - loss: 0.3020 - accuracy:
0.9342 - val_loss: 0.9255 - val_accuracy: 0.8080
Epoch 10/20
16/16 [==============================] - 0s 22ms/step - loss: 0.2520 - accuracy:
0.9441 - val_loss: 0.9158 - val_accuracy: 0.8160
Epoch 11/20
16/16 [==============================] - 0s 22ms/step - loss: 0.2178 - accuracy:
0.9479 - val_loss: 0.9105 - val_accuracy: 0.8110
Epoch 12/20
16/16 [==============================] - 0s 21ms/step - loss: 0.1936 - accuracy:
0.9498 - val_loss: 0.9347 - val_accuracy: 0.8140
Epoch 13/20
16/16 [==============================] - 0s 22ms/step - loss: 0.1707 - accuracy:
0.9546 - val_loss: 0.9786 - val_accuracy: 0.8090
Epoch 14/20
16/16 [==============================] - 0s 22ms/step - loss: 0.1564 - accuracy:
0.9559 - val_loss: 0.9668 - val_accuracy: 0.8080
Epoch 15/20
16/16 [==============================] - 0s 23ms/step - loss: 0.1432 - accuracy:
```

```
0.9569 - val_loss: 0.9978 - val_accuracy: 0.8160
Epoch 16/20
16/16 [==============================] - 0s 24ms/step - loss: 0.1365 - accuracy:
0.9546 - val_loss: 1.0597 - val_accuracy: 0.8060
Epoch 17/20
16/16 [==============================] - 0s 23ms/step - loss: 0.1267 - accuracy:
0.9568 - val_loss: 1.0350 - val_accuracy: 0.8030
Epoch 18/20
16/16 [==============================] - 0s 23ms/step - loss: 0.1267 - accuracy:
0.9564 - val_loss: 1.1126 - val_accuracy: 0.7960
Epoch 19/20
16/16 [==============================] - 0s 22ms/step - loss: 0.1153 - accuracy:
0.9583 - val_loss: 1.0951 - val_accuracy: 0.8010
Epoch 20/20
16/16 [==============================] - 0s 20ms/step - loss: 0.1140 - accuracy:
0.9575 - val_loss: 1.0816 - val_accuracy: 0.8110
71/71 [==============================] - 0s 2ms/step - loss: 1.1830 - accuracy: 0.
7872
최종 평가(loss, accuracy) :  [1.1830408573150635, 0.7871772050857544]
```

46차원보다 훨씬 작은 중간층(예를 들어 4차원)을 두면,

In [35]:

```
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))

results = model.evaluate(X_test, y_test)
print("최종 평가(loss, accuracy) : ", results)
```

```
Epoch 1/20
16/16 [==============================] - 1s 32ms/step - loss: 3.6325 - accuracy: 0.3
187 - val_loss: 3.4300 - val_accuracy: 0.3550
Epoch 2/20
16/16 [==============================] - 0s 22ms/step - loss: 3.2705 - accuracy: 0.3
614 - val_loss: 3.1345 - val_accuracy: 0.3730
Epoch 3/20
16/16 [==============================] - 0s 22ms/step - loss: 2.9529 - accuracy: 0.3
772 - val_loss: 2.8516 - val_accuracy: 0.3840
Epoch 4/20
16/16 [==============================] - 0s 22ms/step - loss: 2.6568 - accuracy: 0.3
893 - val_loss: 2.6141 - val_accuracy: 0.3970
Epoch 5/20
16/16 [==============================] - 0s 22ms/step - loss: 2.4164 - accuracy: 0.4
064 - val_loss: 2.4157 - val_accuracy: 0.4110
Epoch 6/20
16/16 [==============================] - 0s 22ms/step - loss: 2.1864 - accuracy: 0.4
465 - val_loss: 2.2388 - val_accuracy: 0.4410
Epoch 7/20
16/16 [==============================] - 0s 22ms/step - loss: 1.9912 - accuracy: 0.4
682 - val_loss: 2.0951 - val_accuracy: 0.4410
Epoch 8/20
16/16 [==============================] - 0s 22ms/step - loss: 1.8167 - accuracy: 0.4
751 - val_loss: 1.9506 - val_accuracy: 0.4450
Epoch 9/20
16/16 [==============================] - 0s 22ms/step - loss: 1.6398 - accuracy: 0.5
420 - val_loss: 1.8116 - val_accuracy: 0.5900
Epoch 10/20
16/16 [==============================] - 0s 23ms/step - loss: 1.4703 - accuracy: 0.6
662 - val_loss: 1.6910 - val_accuracy: 0.6300
Epoch 11/20
16/16 [==============================] - 0s 24ms/step - loss: 1.3400 - accuracy: 0.6
755 - val_loss: 1.6136 - val_accuracy: 0.6280
Epoch 12/20
16/16 [==============================] - 0s 25ms/step - loss: 1.2507 - accuracy: 0.6
798 - val_loss: 1.5649 - val_accuracy: 0.6370
Epoch 13/20
16/16 [==============================] - 0s 25ms/step - loss: 1.1837 - accuracy: 0.6
901 - val_loss: 1.5378 - val_accuracy: 0.6360
Epoch 14/20
16/16 [==============================] - 0s 25ms/step - loss: 1.1265 - accuracy: 0.7
056 - val_loss: 1.5293 - val_accuracy: 0.6420
```

```
Epoch 15/20
16/16 [==============================] - 0s 23ms/step - loss: 1.0773 - accuracy: 0.7
176 - val_loss: 1.5202 - val_accuracy: 0.6460
Epoch 16/20
16/16 [==============================] - 0s 24ms/step - loss: 1.0379 - accuracy: 0.7
278 - val_loss: 1.5228 - val_accuracy: 0.6580
Epoch 17/20
16/16 [==============================] - 0s 23ms/step - loss: 1.0001 - accuracy: 0.7
379 - val_loss: 1.5152 - val_accuracy: 0.6560
Epoch 18/20
16/16 [==============================] - 0s 23ms/step - loss: 0.9656 - accuracy: 0.7
417 - val_loss: 1.5185 - val_accuracy: 0.6560
Epoch 19/20
16/16 [==============================] - 0s 23ms/step - loss: 0.9361 - accuracy: 0.7
466 - val_loss: 1.5525 - val_accuracy: 0.6610
Epoch 20/20
16/16 [==============================] - 0s 20ms/step - loss: 0.9090 - accuracy: 0.7
549 - val_loss: 1.5539 - val_accuracy: 0.6610
71/71 [==============================] - 0s 2ms/step - loss: 1.6454 - accuracy: 0.65
00
최종 평가(loss, accuracy) :  [1.645433783531189, 0.6500445008277893]
```

검증 정확도가 감소

- 출력층보다 뉴런의 수가 많이 적을 경우, 검증 정확도의 감소 현상이 생길 수 있음.

- 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.

(추가 공부) 데이터 섞기(np.random.shuffle)

- 데이터 섞기 후, 데이터 확인 결과 약 20% 정도 제외한 결과가 전부 다름

In [36]:

```
import copy

test_labels_copy = copy.copy(test_labels)
np.random.shuffle(test_labels_copy)       # 데이터 섞기

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

Out[36]:

```
0.18699910952804988
```

Summary

- 단일 레이블, 다중 분류 문제에서는 N개의 클래스에 대한 확률 분포를 출력하기 위해 **softmax 활성화 함수**를 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