10장 케라스를 사용한 인공 신경망 소개

인공 신경망과 케라스를 이용한 구현 방법

10.1 생물학적 뉴런에서 인공 뉴런까지(1)

- 인공 신경망은 우리 생활에 훨씬 커다란 영향을 미칠 것임
  - 신경망을 훈련하기 위한 데이터가 엄청나게 많아짐
  - 1990년대 이후 **컴퓨터 하드웨어** 가 크게 발전
  - 훈련 알고리즘이 향상
  - 일부 인공 신경망의 이론상 제한이 실전에서는 문제가 되지 않는다고 밝혀짐
  - 인공 신경망이 투자와 진보의 선순환에 진입
  - 2010 스마트폰 출현 -> 데이터가 쌓임

10.1 생물학적 뉴런에서 인공 뉴런까지(2)

10.1.1 생물학적 뉴런

- 생물학적 신경망(biological neural network, BNN) 구조 연구가 활발히 진행

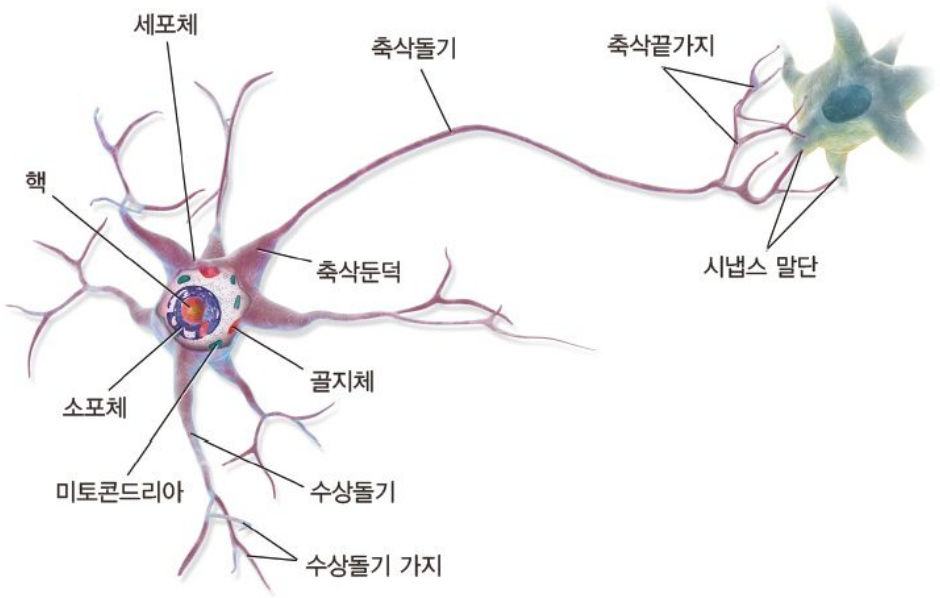

그림 10-1 생물학적 뉴런

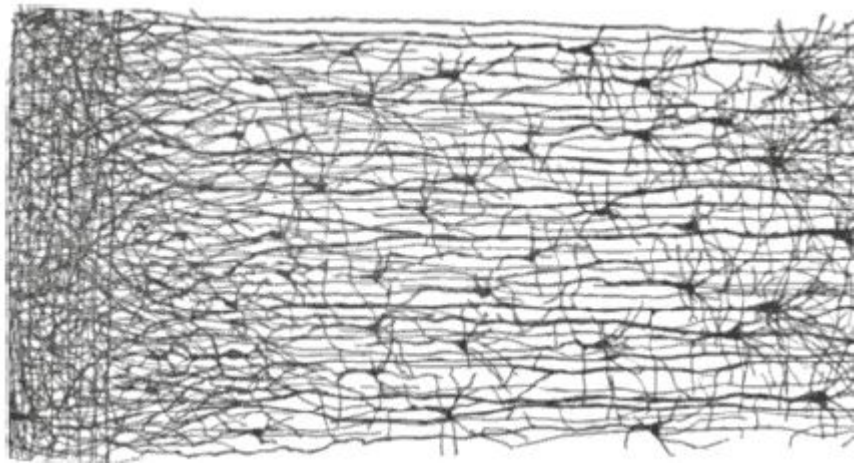

그림 10-2 생물학적 신경망의 여러 층(사람의 피질

10.1 생물학적 뉴런에서 인공 뉴런까지(3)

10.1.2 뉴런을 사용한 논리 연산

- 인공 뉴런(artificial neuron)
  - 매컬러와 피츠가 생물학적 뉴런에서 착안한 매우 단순한 신경망 모델

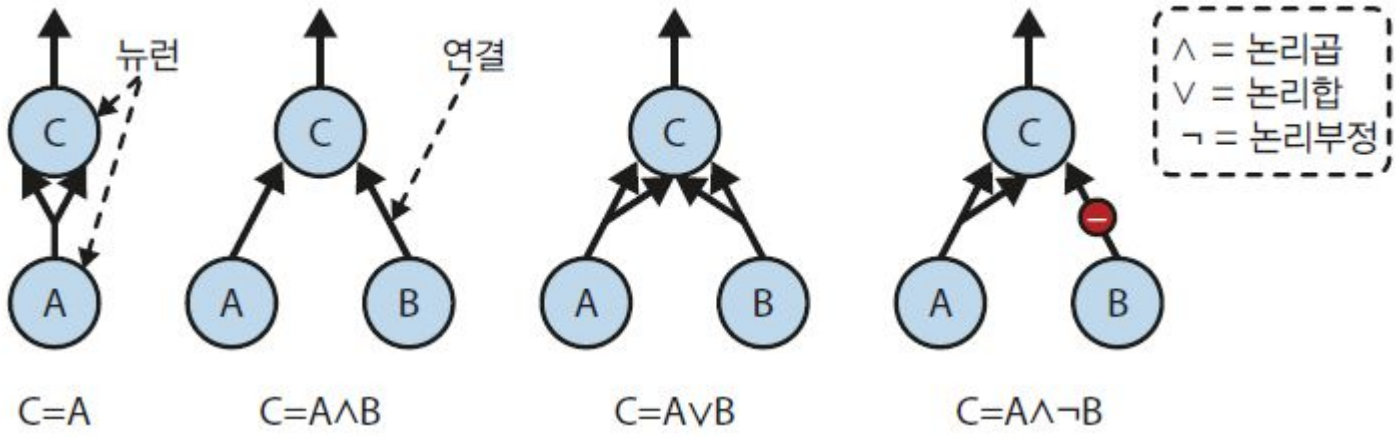

그림 10-3 간단한 논리 연산을 수행하는 인공 신경망

10.1 생물학적 뉴런에서 인공 뉴런까지(4)

10.1.3 퍼셉트론

- 퍼셉트론(perceptron)
  - 가장 간단한 인공 신경망 구조로, 1957년에 프랑크 로젠블라트(Frank Rosenblatt)가 제안
  - 헤비사이드 계단 함수

식 10-1 퍼셉트론에서 일반적으로 사용되는 계단 함수(임곗값을 0으로 가정)

$$\text{heaviside}(z) = \begin{cases} 0 & z < 0\text{일 때} \\ 1 & z \geq 0\text{일 때} \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1 & z < 0\text{일 때} \\ 0 & z = 0\text{일 때} \\ +1 & z > 0\text{일 때} \end{cases}$$

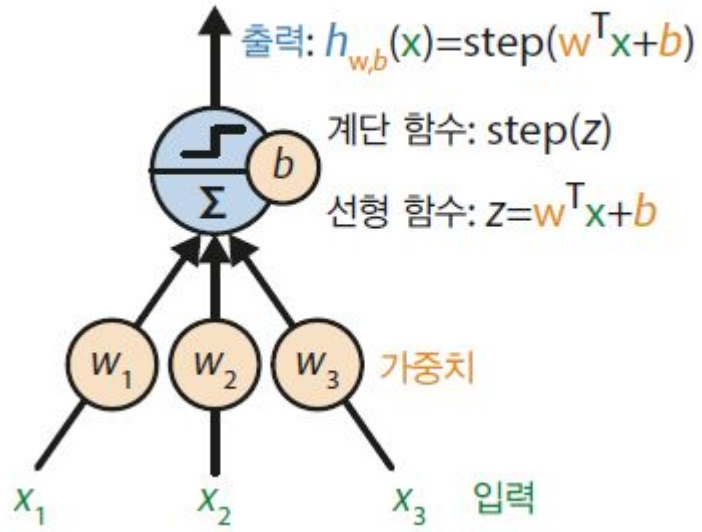

그림 10-4 TLU: 입력의 가중치 합 wTx를 계산하고 편향 b를 더한 다음 계단 함수를 적용하는 인공 뉴런

10.1 생물학적 뉴런에서 인공 뉴런까지(5)

- 퍼셉트론은 하나의 층 안에 놓인 하나 이상의 TLU로 구성되며, 각각의 TLU는 모든 입력에 연결
  - **완전 연결 층**(fully connected layer) 또는 **밀집 층**(dense layer)
  - 입력은 **입력 층**(input layer) 을 구성
  - TLU의 층이 최종 출력을 생성하기 때문에 이를 **출력 층**(output layer) 이라고 함

식 **10-2** 완전 연결 층의 출력 계산

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$$

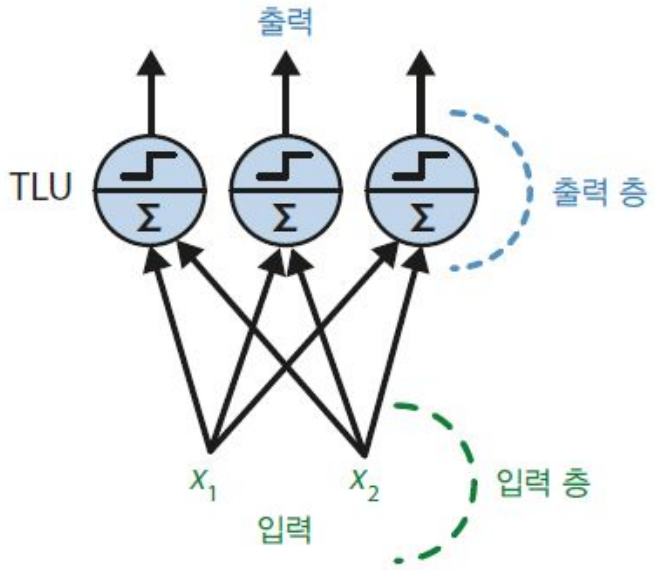

그림 **10-5** 두 개의 입력과 세 개의 출력 뉴런으로 구성된 퍼셉트론의 구조

10.1 생물학적 뉴런에서 인공 뉴런까지(6)

- 헤브의 규칙(Hebb's rule) 또는 헤브의 학습
  - '서로 활성화되는 세포가 서로 연결된다.'
    -즉, 두 뉴런이 동시에 활성화될 때마다 이들 사이의 연결 가중치가 증가하는 경향

식 10-3 퍼셉트론 학습 규칙(가중치 업데이트)

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(y_j - \hat{y}_j\right)x_i$$

- 퍼셉트론 수렴 이론
  - 훈련 샘플이 선형적으로 구분될 수 있다면 이 알고리즘이 정답에 수렴

10.1 생물학적 뉴런에서 인공 뉴런까지(7)

- 사이킷런 Perceptron 클래스

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = (iris.target == 0) # Iris-setosa

per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

X_new = [[2, 0.5], [3, 1]]
y_pred = per_clf.predict(X_new) # 이 두 꽃에 대한 예측은 True와 False입니다.
```

10.1 생물학적 뉴런에서 인공 뉴런까지(8)

- 다층 퍼셉트론(MLP)
  - 퍼셉트론을 여러 개 쌓아올리면 일부 제약을 줄일 수 있음
  - 다층 퍼셉트론의 XOR 문제 풀기

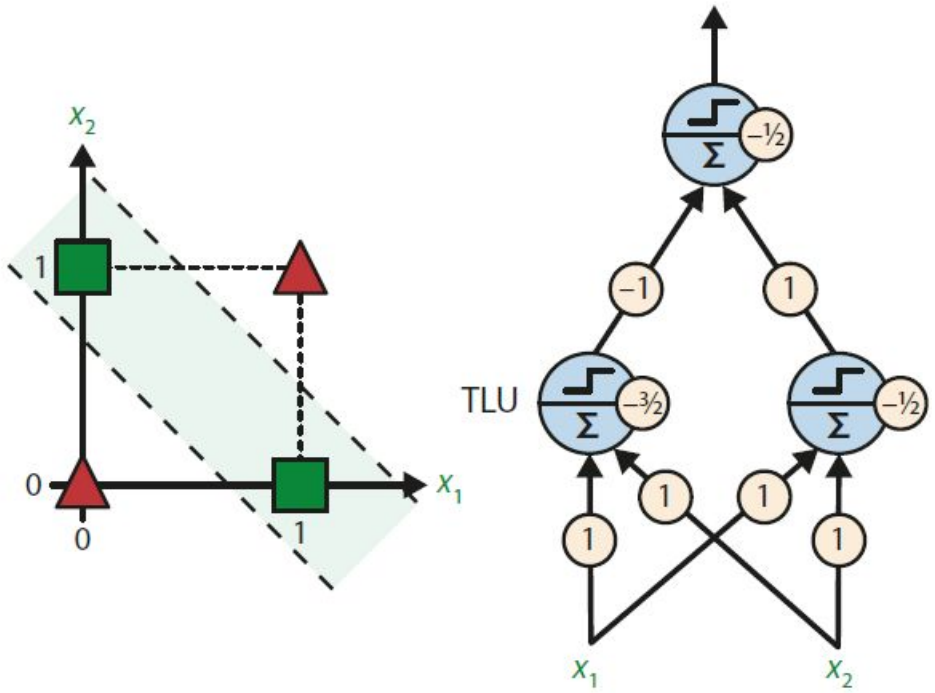

그림 10-6 XOR 분류 문제와 이를 푸는 다층 퍼셉트론

10.1 생물학적 뉴런에서 인공 뉴런까지(9)

10.1.4 다층 퍼셉트론과 역전파

- 다층 퍼셉트론은 입력 층 하나와 은닉 층(hidden layer)이라 불리는 하나 이상의 TLU 층과 마지막 출력 층으로 구성
  - 입력 층과 가까운 층은 보통 하위 층(lower layer), 출력에 가까운 층은 상위 층(upper layer)

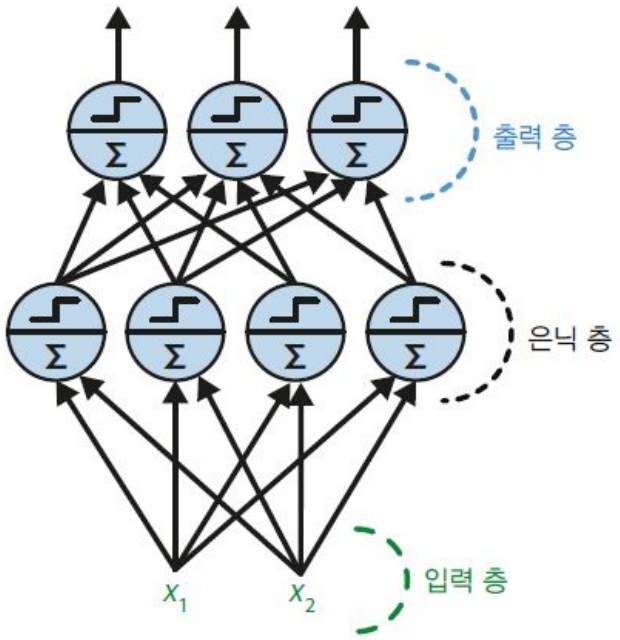

그림 10-7 두 개의 입력, 네 개의 뉴런을 가진 은닉 층, 세 개의 출력 뉴런으로 구성된 다층 퍼셉트론의 구조

10.1 생물학적 뉴런에서 인공 뉴런까지(10)

- 심층 신경망(deep neural network, DNN)
- 후진 모드 자동 미분(reverse-mode automatic differentiation, 또는 reverse-mode autodiff)
  - 모든 그레이디언트를 자동으로 효율적으로 계산하는 알고리즘
- 역전파(backpropagation 또는 backprop)
  - 후진 모드 자동 미분과 경사 하강법을 결합
  - 하나의 미니배치씩 진행하여 전체 훈련 세트를 처리하고 이 과정을 여러 번 반복(에포크)
  - 정방향 계산(forward pass)
  - 네트워크의 출력 오차를 측정하고 기여도 계산(연쇄법칙)
  - 경사 하강법을 수행하여 방금 계산한 오차 그레이디언트를 사용해 네트워크에 있는 모든 연결 가중치를 수정

10.1 생물학적 뉴런에서 인공 뉴런까지(11)

- 역전파 알고리즘에 널리 쓰이는 활성화 함수
  - tanh 함수(하이퍼볼릭 탄젠트 함수): $\tanh(z) = 2\sigma(2z) - 1$
  - ReLU 함수: $\mathrm{ReLU}(z) = \max(0, z)$

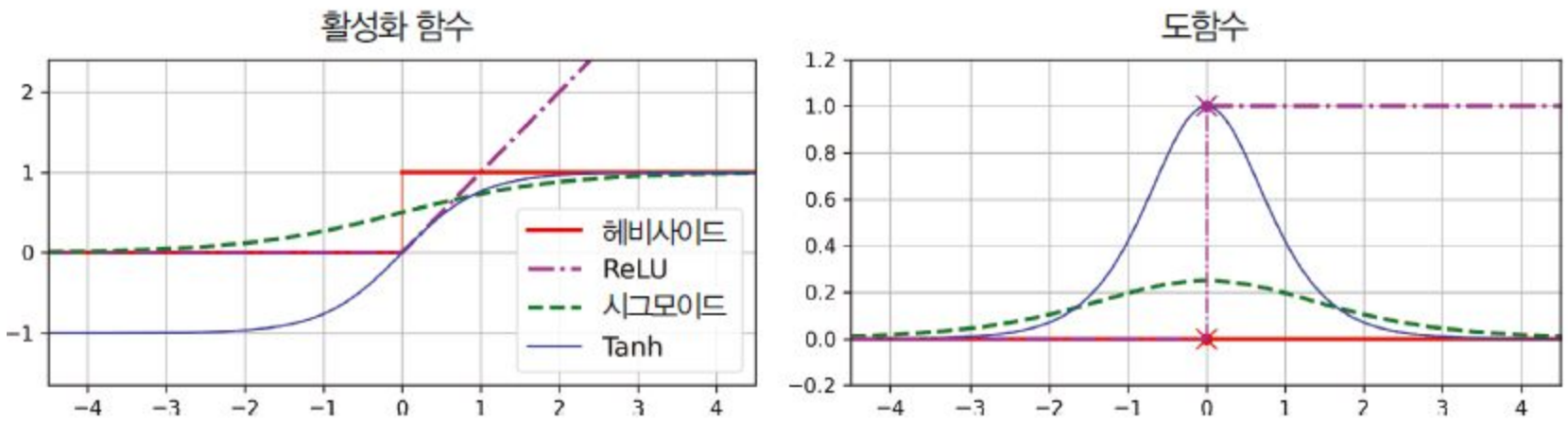

그림 10-8 활성화 함수(왼쪽)와 해당 도함수(오른쪽)

10.1 생물학적 뉴런에서 인공 뉴런까지(12)

10.1.5 회귀를 위한 다층 퍼셉트론

- 사이킷런 MLPRegressor 클래스
  - 각각 50개의 뉴런을 가진 3개의 은닉 층으로 구성된 MLP를 만들고 캘리포니아 주택 데이터셋에서 훈련

```
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)

mlp_reg = MLPRegressor(hidden_layer_sizes=[50, 50, 50], random_state=42)
pipeline = make_pipeline(StandardScaler(), mlp_reg)
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_valid)
rmse = mean_squared_error(y_valid, y_pred, squared=False) # 약 0.505
```

10.1 생물학적 뉴런에서 인공 뉴런까지(13)

- MLPRegressor 클래스는 출력 층에서 활성화 함수를 지원하지 않음
  - MLPRegressor 클래스는 일반적으로 회귀에 필요한 평균 제곱 오차를 사용하지만, 훈련 세트에 이상치가 많은 경우 평균 절대 오차를 대신 사용하는 것이 더 좋을 수 있음
  - 또는 이 두 가지를 조합한 후버 손실(Huber loss)을 사용

표 10-1 회귀 MLP의 전형적인 구조

하이퍼파라미터	일반적인 값
은닉 층 수	문제에 따라 다름(일반적으로 1에서 5 사이)
은닉 층의 뉴런 수	문제에 따라 다름(일반적으로 10에서 100 사이)
출력 뉴런 수	예측 차원마다 하나
은닉 층의 활성화 함수	ReLU
출력 층의 활성화 함수	없음. 또는 (출력이 양수일 때) ReLU/소프트플러스나 (출력을 특정 범위로 제한할 때) 로지스틱/tanh를 사용
손실 함수	MSE 또는 (이상치가 있다면) 후버

10.1 생물학적 뉴런에서 인공 뉴런까지(14)

10.1.6 분류를 위한 다층 퍼셉트론

- 다층 퍼셉트론은 다중 레이블 이진 분류 문제를 쉽게 처리(스팸인지 긴급 메일인지 여부)
  - 시그모이드 활성화 함수를 가진 두 출력 뉴런이 필요
    - 첫 번째 뉴런은 이메일이 스팸일 확률을 출력
    - 두 번째 뉴런은 긴급한 메일일 확률을 출력

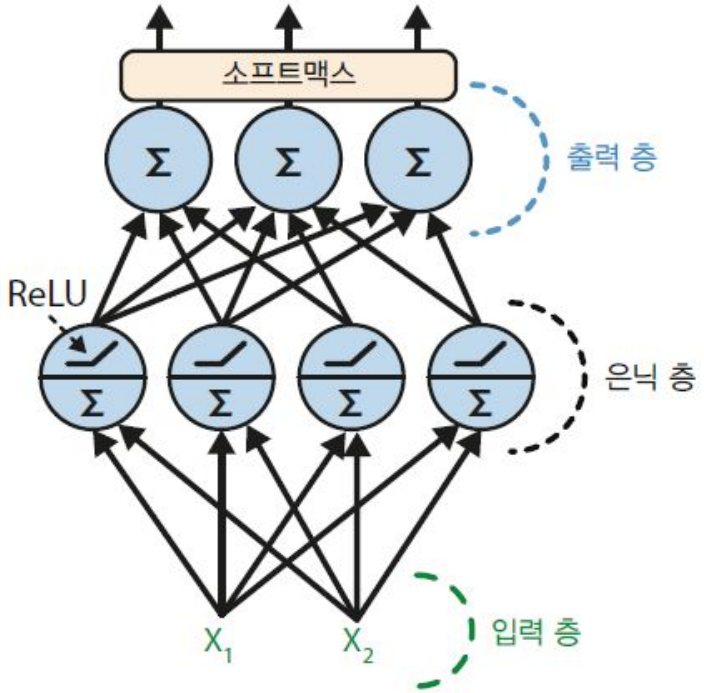

그림 10-9 분류에 사용되는 (ReLU와 소프트맥스를 포함한) 현대적 다층 퍼셉트론

10.1 생물학적 뉴런에서 인공 뉴런까지(15)

- sklearn.neural_network 패키지 아래 MLPClassifier 클래스
  - MLPRegressor 클래스와 거의 동일하지만 MSE가 아닌 크로스 엔트로피를 최소화한다는 점이 다름

표 10-2 분류 MLP의 전형적인 구조

하이퍼파라미터	이진 분류	다중 레이블 분류	다중 분류
은닉 층 수	문제에 따라 다름(일반적으로 1에서 5 사이)		
출력 뉴런 수	1개	이진 레이블마다 1개	클래스마다 1개
출력 층의 활성화 함수	시그모이드 함수	시그모이드 함수	소프트맥스 함수
손실 함수	크로스 엔트로피	크로스 엔트로피	크로스 엔트로피

10.2 케라스로 다층 퍼셉트론 구현하기(1)

10.2.1 시퀀셜 API로 이미지 분류기 만들기

케라스로 데이터셋 적재하기

- 패션 MNIST를 로드
  - 이 데이터셋은 이미 뒤섞여 훈련 세트(60,000개 이미지)와 테스트 세트(10,000개 이미지)로 분할되어 있지만, 검증 세트를 위해 훈련 세트의 마지막 5,000개 이미지를 분리

```
import tensorflow as tf

fashion_mnist = tf.keras.datasets.fashion_mnist.load_data()
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist
X_train, y_train = X_train_full[:-5000], y_train_full[:-5000]
X_valid, y_valid = X_train_full[-5000:], y_train_full[-5000:]
```

  - 사이킷런 대신 케라스를 사용하여 MNIST나 패션 MNIST 데이터를 적재할 때 중요한 차이점
    - 각 이미지가 784 크기의 1D 배열이 아니라 28×28 크기의 배열
    - 픽셀 강도가 실수(0.0에서 255.0까지)가 아니라 정수(0에서 255까지)로 표현

```
>>> X_train.shape
(55000, 28, 28)
>>> X_train.dtype
dtype('uint8')
```

10.2 케라스로 다층 퍼셉트론 구현하기(2)

케라스로 데이터셋 적재하기

- 픽셀 강도를 255.0으로 나누어 0~1 사이 범위로 조정

```
X_train, X_valid, X_test = X_train / 255., X_valid / 255., X_test / 255.
```

- MNIST는 레이블에 해당하는 아이템을 나타내기 위해 클래스 이름의 리스트를 생성

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

  - 예를 들어 훈련 세트에 있는 첫 번째 이미지는 앵클 부츠

```
>>> class_names[y_train[0]]
'Ankle boot'
```

10.2 케라스로 다층 퍼셉트론 구현하기(3)

케라스로 데이터셋 적재하기

- [그림 10-10]은 패션 MNIST 데이터셋의 일부 샘플

그림 10-10 패션 MNIST의 샘플

10.2 케라스로 다층 퍼셉트론 구현하기(4)

시퀀셜 API로 모델 만들기

- 두 개의 은닉 층으로 이루어진 분류용 다층 퍼셉트론

```
tf.random.set_seed(42)  ❶
model = tf.keras.Sequential()  ❷
model.add(tf.keras.layers.Input(shape=[28, 28]))  ❸
model.add(tf.keras.layers.Flatten())  ❹
model.add(tf.keras.layers.Dense(300, activation="relu"))    ❺
model.add(tf.keras.layers.Dense(100, activation="relu"))    ❻
model.add(tf.keras.layers.Dense(10, activation="softmax"))  ❼
```

❶ 결과를 재현할 수 있도록 텐서플로의 랜덤 시드를 설정
❷ S equential 모델을 생성
❸ 첫 번째 층(Input 층)을 만들어 모델에 추가
❹ Flatten 층을 추가
❺ 뉴런 300개를 가진 Dense 은닉 층을 추가
❻ 뉴런 100개를 가진 두 번째 Dense 은닉 층을 추가
❼ 마지막으로 (클래스마다 하나씩) 뉴런 10개를 가진 Dense 출력 층을 추가

10.2 케라스로 다층 퍼셉트론 구현하기(5)

시퀀셜 API로 모델 만들기

- 층을 하나씩 추가하지 않고 Sequential 모델을 만들 때 층의 리스트를 전달
  - 또한 Input 층 대신에 첫 번째 층에 input_shape을 지정

```
model = tf.keras.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

10.2 케라스로 다층 퍼셉트론 구현하기(6)

시퀀셜 API로 모델 만들기

- 모델의 summary() 메서드는 모델에 있는 모든 층을 출력

```
>>> model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0

dense (Dense)                (None, 300)               235500

dense_1 (Dense)              (None, 100)               30100

dense_2 (Dense)              (None, 10)                1010

=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

10.2 케라스로 다층 퍼셉트론 구현하기(7)

시퀀셜 API로 모델 만들기

- layers 속성을 사용하여 모델의 층 목록을 쉽게 얻을 수 있음
  - get_layer() 메서드를 사용하여 층 이름으로 층을 선택

```
>>> model.layers
[<tensorflow.python.keras.layers.core.Flatten at 0x132414e48>,
 <tensorflow.python.keras.layers.core.Dense at 0x1324149b0>,
 <tensorflow.python.keras.layers.core.Dense at 0x1356ba8d0>,
 <tensorflow.python.keras.layers.core.Dense at 0x13240d240>]
>>> hidden1 = model.layers[1]
>>> hidden1.name
'dense'
>>> model.get_layer('dense') is hidden1
True
```

10.2 케라스로 다층 퍼셉트론 구현하기(8)

시퀀셜 API로 모델 만들기

- 층의 모든 파라미터는 get_weights() 메서드와 set_weights() 메서드를 사용해 접근 가능
  - Dense 층의 경우 연결 가중치와 편향이 모두 포함

```
>>> weights, biases = hidden1.get_weights()
>>> weights
array([[ 0.02448617, -0.00877795, -0.02189048, ...,  0.03859074, -0.06889391],
       [ 0.00476504, -0.03105379, -0.0586676 , ..., -0.02763776, -0.04165364],
       ...,
       [ 0.07061854, -0.06960931,  0.07038955, ..., 0.00034875,  0.02878492],
       [-0.06022581,  0.01577859, -0.02585464, ..., 0.00272203, -0.06793761]],
       dtype=float32)
>>> weights.shape
(784, 300)
>>> biases
array([0., 0., 0., 0., 0., 0., 0., 0., 0., ...,  0., 0., 0.], dtype=float32)
>>> biases.shape
(300,)
```

10.2 케라스로 다층 퍼셉트론 구현하기(9)

모델 컴파일

- 모델을 만들고 나서 compile() 메서드를 호출하여 사용할 손실 함수와 옵티마이저(optimizer)를 지정
  - 레이블이 정수 하나로 이루어져 있고 클래스가 배타적이므로 "sparse_categorical_crossentropy" 손실을 사용
  - 만약 샘플마다 클래스별 타깃 확률을 가지고 있다면 "categorical_crossentropy" 손실을 사용
  - 이진 분류나 다중 레이블 이진 분류를 수행한다면 출력 층에 "softmax" 함수 대신 "sigmoid"함수를 사용하고 "binary_crossentropy" 손실을 사용
  - 분류기이므로 훈련과 평가 시에 정확도를 측정하는 것이 유용하므로 metrics=["accuracy"]로 지정

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

10.2 케라스로 다층 퍼셉트론 구현하기(10)

모델 훈련과 평가

- 모델을 훈련을 위해 fit() 메서드를 호출

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                     validation_data=(X_valid, y_valid))
...
Epoch 1/30
1719/1719 [==============================] - 2s 989us/step
  - loss: 0.7220 - sparse_categorical_accuracy: 0.7649
  - val_loss: 0.4959 - val_sparse_categorical_accuracy: 0.8332
Epoch 2/30
1719/1719 [==============================] - 2s 964us/step
  - loss: 0.4825 - sparse_categorical_accuracy: 0.8332
  - val_loss: 0.4567 - val_sparse_categorical_accuracy: 0.8384
[...]
Epoch 30/30
1719/1719 [==============================] - 2s 963us/step
  - loss: 0.2235 - sparse_categorical_accuracy: 0.9200
  - val_loss: 0.3056 - val_sparse_categorical_accuracy: 0.8894
```

10.2 케라스로 다층 퍼셉트론 구현하기(11)

모델 훈련과 평가

- fit() 메서드가 반환하는 History 객체에는 훈련 파라미터(history.params), 수행된 에포크 리스트(history.epoch)가 포함

```
import matplotlib.pyplot as plt
import pandas as pd

pd.DataFrame(history.history).plot(
    figsize=(8, 5), xlim=[0, 29], ylim=[0, 1], grid=True, xlabel="에포크",
    style=["r--", "r--.", "b-", "b-*"])
plt.show()
```

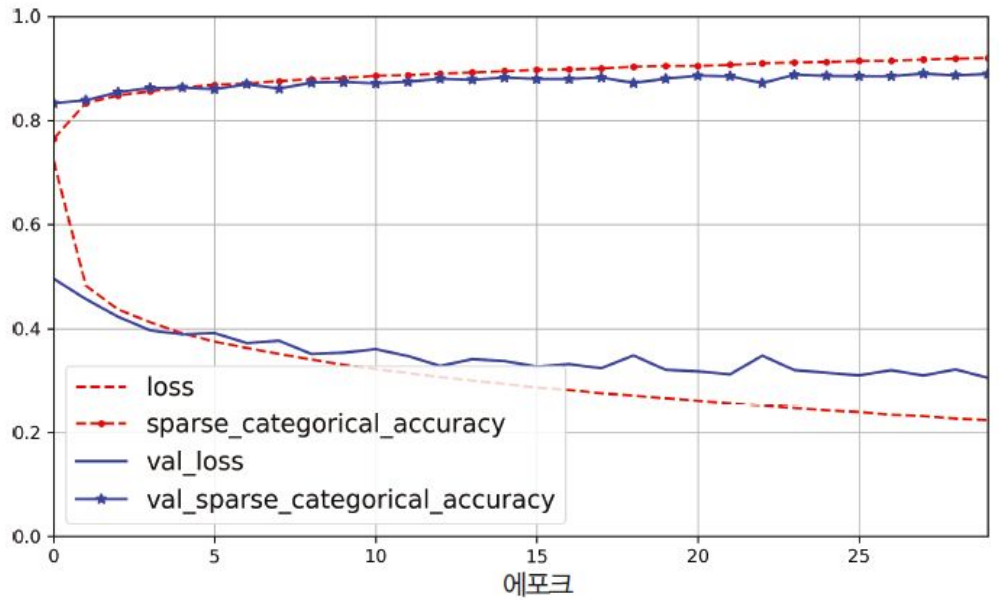

그림 10-11 학습 곡선: 에포크마다 측정한 평균적인 훈련 손실과 정확도 및 에포크의 종료 시점마다 측정한 평균적인 검증 손실과 정확도

10.2 케라스로 다층 퍼셉트론 구현하기(12)

모델 훈련과 평가

- 모델의 검증 정확도가 만족스럽다면 모델을 제품 환경으로 배포하기 전에 테스트 세트로 모델을 평가하여 일반화 오차를 추정
  - evaluate() 메서드를 사용

```
>>> model.evaluate(X_test, y_test)
313/313 [==============================] - 0s 626us/step
  - loss: 0.3243 - sparse_categorical_accuracy: 0.8864
[0.32431697845458984, 0.8863999843597412]
```

10.2 케라스로 다층 퍼셉트론 구현하기(13)

모델로 예측 만들기

- 모델의 predict() 메서드를 사용해 새로운 샘플에 대해 예측

```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.01, 0.  , 0.02, 0.  , 0.97],
       [0.  , 0.  , 0.99, 0.  , 0.01, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

- argmax() 메서드를 사용하여 각 샘플에 대해 가장 높은 확률의 클래스 인덱스를 도출

```
>>> import numpy as np
>>> y_pred = y_proba.argmax(axis=-1)
>>> y_pred
array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

10.2 케라스로 다층 퍼셉트론 구현하기(14)

모델로 예측 만들기

- 분류기는 세 개의 이미지 모두 올바르게 분류

```
>>> y_new = y_test[:3]
>>> y_new
array([9, 2, 1], dtype=uint8)
```

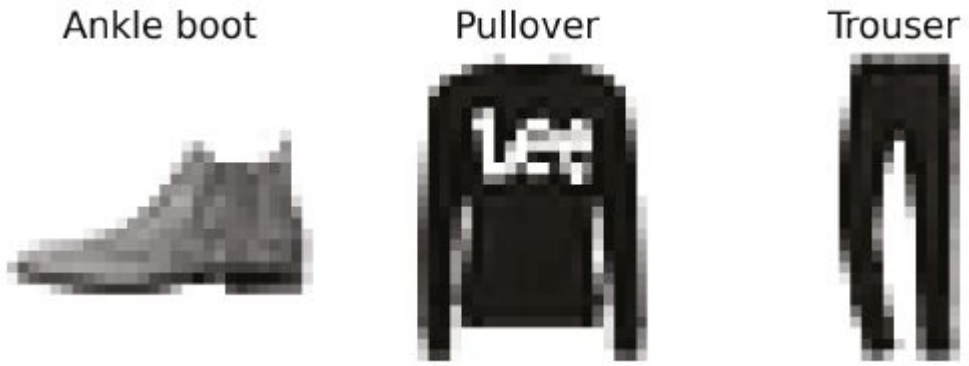

그림 10-12 올바르게 분류된 패션 MNIST 이미지

10.2 케라스로 다층 퍼셉트론 구현하기(15)

10.2.2 시퀀셜 API로 회귀용 다층 퍼셉트론 만들기

- 케라스를 사용하여 구축
  - 캘리포니아 주택 가격 문제로 돌아가서 이전과 동일하게 각각 50개의 뉴런으로 구성된 3개의 은닉 층을 가진 MLP를 사용

```
tf.random.set_seed(42)
norm_layer = tf.keras.layers.Normalization(input_shape=X_train.shape[1:])
model = tf.keras.Sequential([
    norm_layer,
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(1)
])
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])
norm_layer.adapt(X_train)
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
mse_test, rmse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3]
y_pred = model.predict(X_new)
```

10.2 케라스로 다층 퍼셉트론 구현하기(16)

10.2.3 함수형 API로 복잡한 모델 만들기

- 순차적이지 않은 와이드 & 딥(Wide & Deep) 신경망

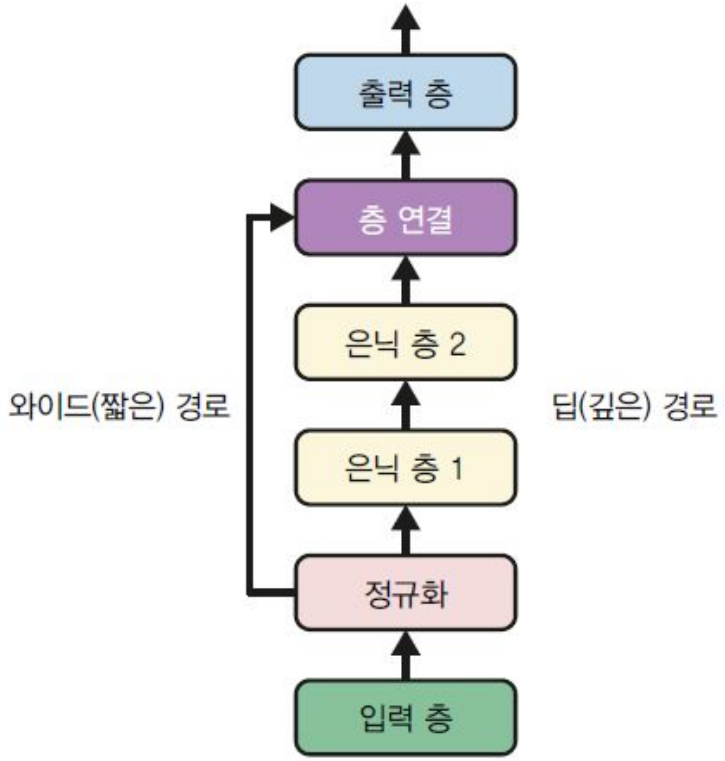

그림 10-13 와이드 & 딥 신경망의 와이드(짧은) 경로와 딥(깊은) 경로

10.2 케라스로 다층 퍼셉트론 구현하기(17)

- 와이드 & 딥(Wide & Deep) 신경망을 캘리포니아 주택 문제에 적용

```
normalization_layer = tf.keras.layers.Normalization()
hidden_layer1 = tf.keras.layers.Dense(30, activation="relu")
hidden_layer2 = tf.keras.layers.Dense(30, activation="relu")
concat_layer = tf.keras.layers.Concatenate()
output_layer = tf.keras.layers.Dense(1)

input_ = tf.keras.layers.Input(shape=X_train.shape[1:])
normalized = normalization_layer(input_)
hidden1 = hidden_layer1(normalized)
hidden2 = hidden_layer2(hidden1)
concat = concat_layer([normalized, hidden2])
output = output_layer(concat)

model = tf.keras.Model(inputs=[input_], outputs=[output])
```

❶ 먼저 입력을 표준화하기 위한 Normalization 층, ReLU 활성화 함수를 사용하여 각각 30개의 뉴런이 있는 두 개의 Dense 층, Concatenate 층, 그리고 출력 층을 위해 활성화 함수 없이 하나의 뉴런이 있는 Dense 층 등 5개의 층을 만들기

❷ Input 객체를 생성

❸ Normalization 층을 함수처럼 사용하여 Input 객체를 전달

❹ 같은 방식으로 normalized를 hidden_layer1에 전달하면 hidden1이 출력되고, hidden1을 hidden_layer2에 전달하면 hidden2가 출력

❺ 지금까지는 층을 순차적으로 연결했지만 concat_layer를 사용하여 입력과 두 번째 은닉 층의 출력을 연결

❻ concat을 output_layer로 전달하여 최종 output을 도출

10.2 케라스로 다층 퍼셉트론 구현하기(18)

- 일부 특성을 짧은 경로로 전달하고 다른 특성들을(중복될 수 있음) 깊은 경로로 전달

```
input_wide = tf.keras.layers.Input(shape=[5]) # 특성 인덱스 0부터 4까지
input_deep = tf.keras.layers.Input(shape=[6]) # 특성 인덱스 2부터 7까지
norm_layer_wide = tf.keras.layers.Normalization()
norm_layer_deep = tf.keras.layers.Normalization()
norm_wide = norm_layer_wide(input_wide)
norm_deep = norm_layer_deep(input_deep)
hidden1 = tf.keras.layers.Dense(30, activation="relu")(norm_deep)
hidden2 = tf.keras.layers.Dense(30, activation="relu")(hidden1)
concat = tf.keras.layers.concatenate([norm_wide, hidden2])
output = tf.keras.layers.Dense(1)(concat)
model = tf.keras.Model(inputs=[input_wide, input_deep], outputs=[output])
```

- 모든 Dense 층은 한 줄에서 생성되어 호출
  - 하지만 Normalization 층의 경우에는 이렇게 할 수 없음
  - 모델을 훈련하기 전에 Normalization 층의 adapt() 메서드를 호출할 수 있도록 이 층에 대한 참조가 필요하기 때문임
- tf.keras.layers.concatenate()을 사용하여 Concatenate 층을 만들고 주어진 입력으로 이 층을 호출
- 두 개의 입력이 있으므로 모델을 만들 때 inputs=[input_wide, input_deep]과 같이 지정

10.2 케라스로 다층 퍼셉트론 구현하기(19)

- [그림 10-14] 여러 개의 입력 다루기

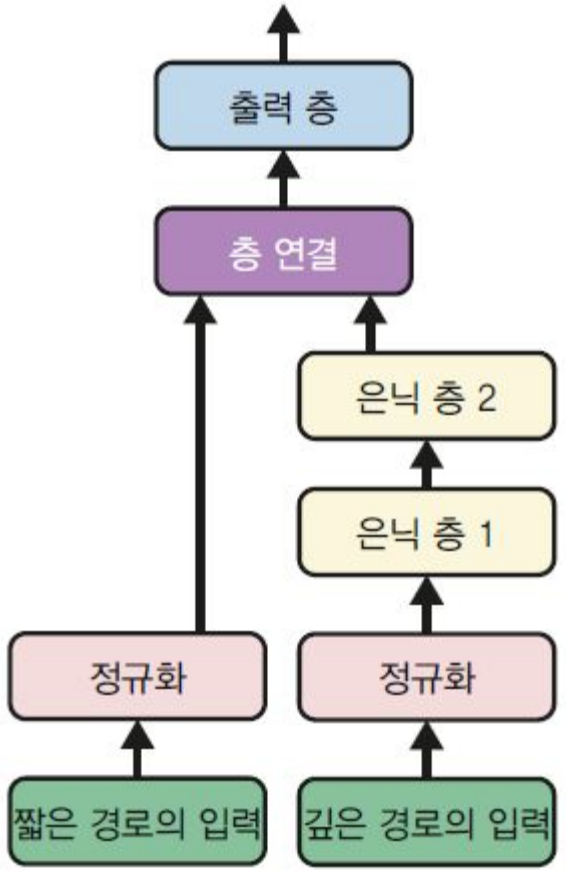

그림 10-14 여러 개의 입력 다루기

10.2 케라스로 다층 퍼셉트론 구현하기(20)

- fit() 메서드를 호출할 때 하나의 입력 행렬 X_train을 전달하는 것이 아니라 입력마다 하나씩 행렬의 튜플 (X_train_wide, X_train_deep)을 전달
  - X_valid에도 동일하게 적용
  - evaluate()나 predict()를 호출할 때 X_test와 X_new에도 동일

```
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])

X_train_wide, X_train_deep = X_train[:, :5], X_train[:, 2:]
X_valid_wide, X_valid_deep = X_valid[:, :5], X_valid[:, 2:]
X_test_wide, X_test_deep = X_test[:, :5], X_test[:, 2:]
X_new_wide, X_new_deep = X_test_wide[:3], X_test_deep[:3]

norm_layer_wide.adapt(X_train_wide)
norm_layer_deep.adapt(X_train_deep)
history = model.fit((X_train_wide, X_train_deep), y_train, epochs=20,
                    validation_data=((X_valid_wide, X_valid_deep), y_valid))
mse_test = model.evaluate((X_test_wide, X_test_deep), y_test)
y_pred = model.predict((X_new_wide, X_new_deep))
```

10.2 케라스로 다층 퍼셉트론 구현하기(21)

- 여러 개의 출력이 필요한 경우
  - 여러 출력이 필요한 작업
  - 동일한 데이터에서 독립적인 여러 작업을 수행
  - 규제 기법으로 사용

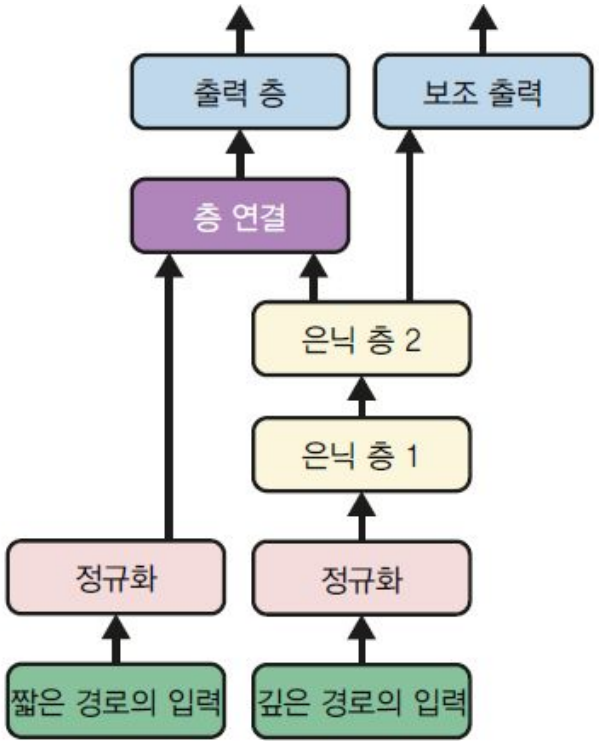

그림 10-15 여러 개의 출력 다루기. 여기에서는 규제를 위해 보조 출력을 추가

10.2 케라스로 다층 퍼셉트론 구현하기(22)

- 보조 출력을 추가
  - 적절한 층에 연결하고 모델의 출력 리스트에 추가

```
[...]  # 출력 층까지는 이전과 동일합니다.
output = tf.keras.layers.Dense(1)(concat)
aux_output = tf.keras.layers.Dense(1)(hidden2)
model = tf.keras.Model(inputs=[input_wide, input_deep],
                       outputs=[output, aux_output])
```

- 주 출력의 손실에 더 많은 가중치를 부여해야 할 경우
  - 모델을 컴파일할 때 손실 가중치를 지정

```
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss=("mse", "mse"), loss_weights=(0.9, 0.1), optimizer=optimizer,
              metrics=["RootMeanSquaredError"])
```

10.2 케라스로 다층 퍼셉트론 구현하기(23)

- 모델을 훈련할 때 각 출력에 대한 레이블을 제공

```
norm_layer_wide.adapt(X_train_wide)
norm_layer_deep.adapt(X_train_deep)
history = model.fit(
    (X_train_wide, X_train_deep), (y_train, y_train), epochs=20,
    validation_data=((X_valid_wide, X_valid_deep), (y_valid, y_valid))
)
```

- 모델을 평가하면 케라스는 개별 손실과 측정 지표는 물론 손실의 가중치 합을 반환

```
eval_results = model.evaluate((X_test_wide, X_test_deep), (y_test, y_test))
weighted_sum_of_losses, main_loss, aux_loss, main_rmse, aux_rmse = eval_results
```

10.2 케라스로 다층 퍼셉트론 구현하기(24)

- predict() 메서드는 각 출력에 대한 예측을 반환

```
y_pred_main, y_pred_aux = model.predict((X_new_wide, X_new_deep))
```

- predict() 메서드는 튜플을 반환하며, 딕셔너리를 반환하기 위한 return_dict 매개변수가 없음
  - model.output_names를 사용하여 딕셔너리를 만들 수 있음

```
y_pred_tuple = model.predict((X_new_wide, X_new_deep))
y_pred = dict(zip(model.output_names, y_pred_tuple))
```

10.2 케라스로 다층 퍼셉트론 구현하기(25)

10.2.4 서브클래싱 API로 동적 모델 만들기

- 시퀀셜 API와 함수형 API는 모두 선언적(declarative)
  - 사용할 층과 연결 방식을 먼저 정의
  - 그 다음 모델에 데이터를 주입하여 훈련이나 추론을 시작할 수 있음
  - 장점
    - 모델을 저장하거나 복사, 공유하기 쉬움
    - 모델의 구조를 출력하거나 분석하기 좋음
    - 프레임워크가 크기를 짐작하고 타입을 확인하여 에러를 (모델에 데이터가 주입되기 전에) 일찍 발견할 수 있음
    - 전체 모델이 층으로 구성된 정적 그래프이므로 디버깅하기도 쉬움
  - 단점
    - 정적 구조
      - 어떤 모델은 반복문을 포함하고 다양한 크기를 다루어야 하며 조건문을 가지는 등 여러 가지 동적인 구조를 필요로 함
  - 명령형(imperative) 프로그래밍 스타일이 필요하다면 서브클래싱 API(subclassing API)가 정답

10.2 케라스로 다층 퍼셉트론 구현하기(26)

- WideAndDeepModel 클래스
  - 앞서 함수형 API로 만든 모델과 동일한 기능을 수행

```
class WideAndDeepModel(tf.keras.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs) # 모델에 이름을 부여하기 위해 필요합니다.
        self.norm_layer_wide = tf.keras.layers.Normalization()
        self.norm_layer_deep = tf.keras.layers.Normalization()
        self.hidden1 = tf.keras.layers.Dense(units, activation=activation)
        self.hidden2 = tf.keras.layers.Dense(units, activation=activation)
        self.main_output = tf.keras.layers.Dense(1)
        self.aux_output = tf.keras.layers.Dense(1)

    def call(self, inputs):
        input_wide, input_deep = inputs
        norm_wide = self.norm_layer_wide(input_wide)
        norm_deep = self.norm_layer_deep(input_deep)
        hidden1 = self.hidden1(norm_deep)
        hidden2 = self.hidden2(hidden1)
        concat = tf.keras.layers.concatenate([norm_wide, hidden2])
        output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return output, aux_output

model = WideAndDeepModel(30, activation="relu", name="my_cool_model")
```

10.2 케라스로 다층 퍼셉트론 구현하기(27)

10.2.5 모델 저장과 복원하기

- 훈련된 케라스 모델을 저장

```
model.save("my_keras_model", save_format="tf")
```

- 모델을 로드
  - save_weights()와 load_weights()를 사용하여 파라미터 값만 저장하고 로드할 수도 있음

```
model = tf.keras.models.load_model("my_keras_model")
y_pred_main, y_pred_aux = model.predict((X_new_wide, X_new_deep))
```

10.2 케라스로 다층 퍼셉트론 구현하기(28)

10.2.6 콜백 사용하기

- ModelCheckpoint는 훈련하는 동안 일정한 간격으로 모델의 체크포인트를 저장
  - 기본적으로 매 에포크의 끝에서 호출

```
checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("my_checkpoints",
                                                   save_weights_only=True)
history = model.fit([...], callbacks=[checkpoint_cb])
```

- EarlyStopping 콜백을 사용
  - 일정 에포크(patience 매개변수로 지정) 동안 검증 세트에 대한 점수가 향상되지 않으면 훈련을 멈춤
  - restore_best_weights=True로 지정하면 훈련이 끝난 후 최상의 모델을 복원

```
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=10,
                                                     restore_best_weights=True)
history = model.fit([...], callbacks=[checkpoint_cb, early_stopping_cb])
```

10.2 케라스로 다층 퍼셉트론 구현하기(29)

- 사용자 정의 콜백 만들기
  - 예) 훈련하는 동안 검증 손실과 훈련 손실의 비율을 출력(즉, 과대적합을 감지)

```
class PrintValTrainRatioCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        ratio = logs["val_loss"] / logs["loss"]
        print(f"Epoch={epoch}, val/train={ratio:.2f}")
```

10.2 케라스로 다층 퍼셉트론 구현하기(30)

10.2.7 텐서보드로 시각화하기

- 텐서보드는 매우 좋은 인터랙티브 시각화 도구
  - 훈련하는 동안 학습 곡선을 그리거나 여러 실행 간의 학습 곡선을 비교하고 계산 그래프 시각화와 훈련 통계 분석을 수행
  - 모델이 생성한 이미지를 확인하거나 3D에 투영된 복잡한 다차원 데이터를 시각화하고 자동으로 클러스터링하며 네트워크 프로파일링(속도를 측정하여 병목 현상 파악) 등에 사용
  - 텐서보드는 텐서플로를 설치할 때 자동으로 설치
    - 프로파일링 데이터를 시각화하려면 텐서보드 플러그인이 필요
      https://homl.info/install의 설치 지침에 따라 로컬에서 모든 것을 실행했다면 플러그인이 이미 설치됨
    - 코랩을 사용하는 경우 다음 명령을 실행

```
%pip install -q -U tensorboard-plugin-profile
```

10.2 케라스로 다층 퍼셉트론 구현하기(31)

- 텐서보드를 사용하려면 프로그램을 수정하여 이벤트 파일(event file)이라는 특별한 이진 로그 파일에 시각화하려는 데이터를 출력
  - 루트 로그 디렉터리의 이름을 my_logs로 지정하고, 현재 날짜와 시간을 기준으로 서브디렉터리의 경로를 생성하는 함수를 정의하여 실행할 때마다 다른 경로를 만들기

```
from pathlib import Path
from time import strftime

def get_run_logdir(root_logdir="my_logs"):
    return Path(root_logdir) / strftime("run_%Y_%m_%d_%H_%M_%S")

run_logdir = get_run_logdir() # 예: my_logs/run_2022_08_01_17_25_59
```

- TensorBoard() 콜백
  - 로그 디렉터리(필요한 경우 상위 디렉터리와 함께)를 생성하고, 훈련 중에 이벤트 파일을 만들어 요약 정보를 기록

```
tensorboard_cb = tf.keras.callbacks.TensorBoard(run_logdir,
                                                profile_batch=(100, 200))
history = model.fit([...], callbacks=[tensorboard_cb])
```

10.2 케라스로 다층 퍼셉트론 구현하기(32)

- 학습률을 0.001에서 0.002로 변경하고 코드를 다시 실행
  - 새로운 서브디렉터리가 만들어짐을 확인

```
my_logs
├── run_2022_08_01_17_25_59
|   |   └── plugins
|   |       └── profile
|   |           └── 2022_08_01_17_26_02
|   |               ├── my_host_name.input_pipeline.pb
|   |               └── [...]
|   ├─────── train
|   |       ├── events.out.tfevents.1659331561.my_host_name.42042.0.v2
|   |       ├── events.out.tfevents.1659331562.my_host_name.profile-empty
|   └─────── validation
|           └── events.out.tfevents.1659331562.my_host_name.42042.1.v2
└── run_2022_08_01_17_31_12
    └── [...]
```

  - 텐서보드용 주피터 확장 프로그램을 로드

```
%load_ext tensorboard
%tensorboard --logdir=./my_logs
```

10.2 케라스로 다층 퍼셉트론 구현하기(33)

- 텐서보드의 사용자 인터페이스

그림 10-16 텐서보드로 학습 곡선 시각화하기

10.2 케라스로 다층 퍼셉트론 구현하기(34)

- create_file_writer() 함수를 사용하여 SummaryWriter를 생성하고, 이를 파이썬 콘텍스트로 사용하여 스칼라, 히스토그램, 이미지, 오디오 및 텍스트를 기록

```
test_logdir = get_run_logdir()
writer = tf.summary.create_file_writer(str(test_logdir))
with writer.as_default():
    for step in range(1, 1000 + 1):
        tf.summary.scalar("my_scalar", np.sin(step / 10), step=step)

        data = (np.random.randn(100) + 2) * step / 100      # 점점 커집니다.
        tf.summary.histogram("my_hist", data, buckets=50, step=step)

        images = np.random.rand(2, 32, 32, 3) * step / 1000 # 점점 밝아집니다.
        tf.summary.image("my_images", images, step=step)

        texts = ["The step is " + str(step), "Its square is " + str(step ** 2)]
        tf.summary.text("my_text", texts, step=step)

        sine_wave = tf.math.sin(tf.range(12000) / 48000 * 2 * np.pi * step)
        audio = tf.reshape(tf.cast(sine_wave, tf.float32), [1, -1, 1])
        tf.summary.audio("my_audio", audio, sample_rate=48000, step=step)
```

10.3 신경망 하이퍼파라미터 튜닝하기(1)

- 하이퍼파라미터 튜닝
  - GridSearchCV 또는 RandomizedSearchCV를 사용하여 하이퍼파라미터를 미세 튜닝
    - SciKeras 라이브러리의 KerasRegressor와 KerasClassifier 래퍼 클래스를 사용
  - 케라스 튜너(Keras Tuner) 라이브러리를 사용
- 패션 MNIST 이미지를 분류하기 위한 MLP를 만들고 컴파일
  - 은닉 층의 수(n_hidden), 각 층의 뉴런 수(n_neurons), 학습률(learning_rate), 사용할 옵티마이저(optimizer) 등의 하이퍼파라미터를 사용

```
import keras_tuner as kt

def build_model(hp):
    n_hidden = hp.Int("n_hidden", min_value=0, max_value=8, default=2)
    n_neurons = hp.Int("n_neurons", min_value=16, max_value=256)
    learning_rate = hp.Float("learning_rate", min_value=1e-4, max_value=1e-2,
                             sampling="log")
    optimizer = hp.Choice("optimizer", values=["sgd", "adam"])
    if optimizer == "sgd":
        optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)
    else:
        optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)

    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Flatten())
    for _ in range(n_hidden):
        model.add(tf.keras.layers.Dense(n_neurons, activation="relu"))
    model.add(tf.keras.layers.Dense(10, activation="softmax"))
    model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
                  metrics=["accuracy"])
    return model
```

10.3 신경망 하이퍼파라미터 튜닝하기(2)

- 기본적인 랜덤 서치를 수행
  - kt.RandomSearch 튜너를 만들고 build_model 함수를 생성자에 전달한 후 튜너의 search() 메서드를 호출

```
random_search_tuner = kt.RandomSearch(
    build_model, objective="val_accuracy", max_trials=5, overwrite=True,
    directory="my_fashion_mnist", project_name="my_rnd_search", seed=42)
random_search_tuner.search(X_train, y_train, epochs=10,
                           validation_data=(X_valid, y_valid))
```

- 검색을 완료하면 이에 해당하는 최상의 모델을 획득

```
top3_models = random_search_tuner.get_best_models(num_models=3)
best_model = top3_models[0]
```

10.3 신경망 하이퍼파라미터 튜닝하기(3)

- get_best_hyperparameters()를 호출하여 최상의 모델의 kt.HyperParameters를 얻을 수도 있음

```
>>> top3_params = random_search_tuner.get_best_hyperparameters(num_trials=3)
>>> top3_params[0].values # 최상의 하이퍼파라미터 값
{'n_hidden': 5,
 'n_neurons': 70,
 'learning_rate': 0.00041268008323824807,
 'optimizer': 'adam'}
```

10.3 신경망 하이퍼파라미터 튜닝하기(4)

- 각 튜너는 오라클(oracle)의 안내를 받음
  - 오라클은 모든 시도를 기록하기 때문에 최상의 시도를 요청하여 해당 시도의 요약을 출력

```
>>> best_trial = random_search_tuner.oracle.get_best_trials(num_trials=1)[0]
>>> best_trial.summary()
Trial summary
Hyperparameters:
n_hidden: 5
n_neurons: 70
learning_rate: 0.00041268008323824807
optimizer: adam
Score: 0.8736000061035156
```

10.3 신경망 하이퍼파라미터 튜닝하기(5)

- 최상의 하이퍼파라미터와 검증 정확도가 표시

```
>>> best_trial.metrics.get_last_value("val_accuracy")
0.8736000061035156
```

- 최상의 모델 성능이 만족스럽다면 전체 훈련 세트(X_train_full와 y_train_full)에서 몇 번의 에포크 동안 이어서 훈련한 다음 테스트 세트에서 평가하고 제품에 배포

```
best_model.fit(X_train_full, y_train_full, epochs=10)
test_loss, test_accuracy = best_model.evaluate(X_test, y_test)
```

10.3 신경망 하이퍼파라미터 튜닝하기(6)

- 데이터 전처리 하이퍼파라미터 또는 배치 크기와 같은 model.fit() 매개변수를 미세 튜닝해야 할 경우
  - kt.HyperModel 클래스의 서브클래스를 만들고 build()와 fit() 메서드 두 개를 정의

```
class MyClassificationHyperModel(kt.HyperModel):
    def build(self, hp):
        return build_model(hp)
    def fit(self, hp, model, X, y, **kwargs):
        if hp.Boolean("normalize"):
            norm_layer = tf.keras.layers.Normalization()
            X = norm_layer(X)
        return model.fit(X, y, **kwargs)
```

  - build_model() 함수 대신 이 클래스의 객체를 원하는 튜너에 전달
    - 예) MyClassificationHyperModel 객체를 기반으로 kt.Hyperband 튜너

```
hyperband_tuner = kt.Hyperband(
    MyClassificationHyperModel(), objective="val_accuracy", seed=42,
    max_epochs=10, factor=3, hyperband_iterations=2,
    overwrite=True, directory="my_fashion_mnist", project_name="hyperband")
```

10.3 신경망 하이퍼파라미터 튜닝하기(7)

- 하이퍼밴드 튜너를 실행
  - 루트 로그 디렉터리(튜너가 각 시도마다 다른 서브디렉터리를 사용)를 지정한 텐서보드 콜백과 조기 종료 콜백을 사용

```
root_logdir = Path(hyperband_tuner.project_dir) / "tensorboard"
tensorboard_cb = tf.keras.callbacks.TensorBoard(root_logdir)
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=2)
hyperband_tuner.search(X_train, y_train, epochs=10,
                       validation_data=(X_valid, y_valid),
                       callbacks=[early_stopping_cb, tensorboard_cb])
```

10.3 신경망 하이퍼파라미터 튜닝하기(8)

- kt.BayesianOptimization 튜너
  - 가우스 과정(Gaussian process) 확률 모델을 적용하여 하이퍼파라미터 공간의 어느 영역이 가장 유망한지 점진적으로

```
bayesian_opt_tuner = kt.BayesianOptimization(
    MyClassificationHyperModel(), objective="val_accuracy", seed=42,
    max_trials=10, alpha=1e-4, beta=2.6,
    overwrite=True, directory="my_fashion_mnist", project_name="bayesian_opt")
bayesian_opt_tuner.search([...])
```

10.3 신경망 하이퍼파라미터 튜닝하기(9)

10.3.1 은닉 층 개수

- 복잡한 문제에서는 심층 신경망이 얕은 신경망보다 파라미터 효율성(parameter efficiency)이 훨씬 좋음
  - 실제 데이터는 계층 구조를 가진 경우가 많으므로 심층 신경망은 이런 면에서 유리
    - 아래쪽 은닉 층은 저수준의 구조를 모델링(여러 방향이나 모양의 선)
    - 중간 은닉 층은 저수준의 구조를 연결해 중간 수준의 구조를 모델링(사각형, 원)
    - 가장 위쪽 은닉 층과 출력 층은 중간 수준의 구조를 연결해 고수준의 구조를 모델링(얼굴)
  - 계층 구조는 심층 신경망이 좋은 솔루션으로 빨리 수렴하게 도와줄 뿐만 아니라 새로운 데이터에 일반화되는 능력도 향상
  - 전이 학습(transfer learning)

10.3 신경망 하이퍼파라미터 튜닝하기(10)

10.3.2 은닉 층의 뉴런 개수

- 입력 층과 출력 층의 뉴런 개수는 해당 작업에 필요한 입력과 출력의 형태에 따라 결정
  - 예를 들어 MNIST는 28×28 = 784개의 입력 뉴런과 10개의 출력 뉴런이 필요
- 은닉 층의 구성 방식은 일반적으로 각 층의 뉴런을 점점 줄여서 깔때기처럼 구성
  - 저수준의 많은 특성이 고수준의 적은 특성으로 합쳐질 수 있기 때문임
- 층의 개수와 마찬가지로 네트워크가 과대적합이 시작되기 전까지 점진적으로 뉴런 수를 늘릴 수 있음
  - 필요한 것보다 더 많은 층과 뉴런을 가진 모델을 선택한 다음, 과대적합되지 않도록 조기 종료나 규제 기법을 사용

10.3 신경망 하이퍼파라미터 튜닝하기(11)

10.3.3 학습률, 배치 크기 그리고 다른 하이퍼파라미터

학습률

- 일반적으로 최적의 학습률은 최대 학습률(4장에서 보았듯이 훈련 알고리즘이 발산하는 학습률)의 절반 정도

옵티마이저

- 고전적인 평범한 미니배치 경사 하강법보다 더 좋은 옵티마이저를 선택하는 것(그리고 이 옵티마이저의 하이퍼파라미터를 튜닝하는 것)도 매우 중요

배치 크기

- 배치 크기는 모델 성능과 훈련 시간에 큰 영향
- GPU RAM에 맞는 가장 큰 배치 크기 사용이 권장

활성화 함수

- 일반적으로 ReLU 활성화 함수가 모든 은닉 층에 좋은 기본값
- 출력 층의 활성화 함수는 수행하는 작업에 따라 달라짐

반복 횟수

- 대부분의 경우 훈련 반복 횟수는 튜닝할 필요가 없음. 대신 조기 종료를 사용

연습문제 (1)

1. 텐서플로 플레이그라운드(https://playground.tensorflow.org)는 텐서플로 팀에서 만든 편리한 신경망 시뮬레이터. 이 연습문제에서 클릭 몇 번만으로 이진 분류기 몇 개를 훈련함. 또 모델 구조와 하이퍼파라미터를 조작하여 신경망의 작동 방식과 하이퍼파라미터의 역할에 대해 이해해봄.
   a. 신경망이 학습한 패턴. 실행Run 버튼(왼쪽 상단)을 클릭해 기본 신경망을 훈련. 얼마나 빨리 이 분류 문제에 대한 좋은 솔루션을 찾는지 확인.
   첫 번째 은닉층에 있는 뉴런은 단순한 패턴을 학습. 반면 두 번째 은닉층에 있는 뉴런은 첫 번째 은닉층의 단순한 패턴을 조금 더 복잡한 패턴으로 연결.
   일반적으로 층이 많을수록 더 복잡한 패턴이 만들어짐.
   b. 활성화 함수. tanh 활성화 함수를 ReLU 활성화 함수로 바꾸고 이 신경망을 다시 훈련해보기. 더 빠르게 솔루션을 찾지만 이번에는 선형 경계가 만들어짐. 이는 ReLU 함수의 특성 때문임.
   c. 지역 최솟값의 위험.
   세 개의 뉴런과 하나의 은닉층만 있는 네트워크 구조로 수정하기. 여러 번 훈련 시도(네트워크 가중치를 초기화하려면 리셋 버튼을 누르고 그다음 실행 버튼을 클릭). 훈련에 걸리는 시간에 차이가 나며, 이따금 지역 최솟값에 갇히기도 함

연습문제 (1–2)

d. 신경망이 너무 작으면 어떤 일이 일어날까? 뉴런 한 개를 삭제하고 두 개만 남겨보기. 이제 여러 번 훈련해봐도 이 신경망은 좋은 솔루션을 찾을 수 없음.
이 모델은 파라미터가 너무 적어서 구조적으로 훈련 세트에 과소적합됨.

e. 신경망이 너무 크면 어떤 일이 일어날까? 뉴런의 수를 여덟 개로 늘리고 신경망을 여러 번 훈련해보기. 모두 빠르게 훈련되고 지역 최솟값에 갇히지 않음. 이는 신경망 이론에서 발견된 중요한 한 사실을 알려줌.
대규모 신경망은 거의 절대로 지역 최솟값에 갇히지 않음. 지역 최적점에 도달했더라도 거의 전역 최적점만큼 좋은 솔루션임.
그러나 여전히 긴 평탄한 지역에 오랫 동안 갇힐 수 있음.

f. 심층 신경망에서 그레이디언트 소실 문제.
나선형 데이터셋을 선택('DATA' 항목에 있는 오른쪽 아래 데이터셋). 각각 뉴런을 여덟 개 가진 은닉층 네 개로 네트워크 구조를 바꾸기. 훈련 시간이 더 오래 걸리고 이따금 긴 시간 동안 평탄한 지역에 갇힘.
가장 상위 층(오른쪽 층)에 있는 뉴런이 하위 층(왼쪽 층)에 있는 뉴런보다 더 빨리 학습되는 경향이 있음.

- 이를 '그레이디언트 소실vanishing gradient'이라고 부름. 이 문제는 더 좋은 가중치 초기화와 다른 기법을 사용해 감소시킬 수 있음. (11장에서 소개할) 고급 옵티마이저(AdaGrad나 Adam 등)나 배치 정규화batch normalization임.

g. 더 실험해보기. 다른 하이퍼파라미터를 사용해 한 시간 가량 실험.
각 하이퍼파라미터의 역할을 확인하고 신경망에 대한 이해를 높임.

연습문제 (2)

2. ([그림 10-3]에 있는 것과 같은) 초창기 인공 뉴런을 사용해 A⊕B(⊕는 XOR 연산)를 계산하는 인공 신경망을 그려보기. 힌트: $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$

3. 고전적인 퍼셉트론(즉, 퍼셉트론 훈련 알고리즘으로 훈련된 단일 TLU)보다 로직스틱 회귀 분류기를 일반적으로 선호하는 이유는 무엇인가? 퍼셉트론을 어떻게 수정하면 로지스틱 회귀 분류기와 동등하게 만들 수 있나?

4. 왜 초창기의 다층 퍼셉트론을 훈련할 때 로지스틱 활성화 함수가 핵심 요소였나?

5. 인기 많은 활성화 함수 세 가지는 무엇인가? 이를 그려볼 수 있나?

6. 통과 뉴런 10개로 구성된 입력층, 뉴런 50개로 구성된 은닉층, 뉴런 3개로 구성된 출력층으로 이루어진 다층 퍼셉트론이 있다고 가정. 모든 뉴런은 ReLU 활성화 함수를 사용.
   a. 입력 행렬 X의 크기는?
   b. 은닉층의 가중치 벡터 Wh와 편향 벡터 bh의 크기는?
   c. 출력층의 가중치 벡터 Wo와 편향 벡터 bo의 크기는?
   d. 네트워크의 출력 행렬 Y의 크기는?
   e. X, Wh, bh, Wo, bo의 함수로 네트워크의 출력 행렬 Y를 계산하는 식은?

연습문제 (3)

7. 스팸 메일을 분류하기 위해서는 출력층에 몇 개의 뉴런이 필요할까?
   출력층에 어떤 활성화 함수를 사용해야 할까?
   MNIST 문제라면 출력층에 어떤 활성화 함수를 사용하고 뉴런은 몇 개가 필요할까?
   2장에서 본 주택 가격 예측용 네트워크에 대해 같은 질문의 답을 찾아보기

8. 역전파란 무엇이고 어떻게 작동하나? 역전파와 후진 모드 자동 미분의 차이점은 무엇인가?

9. 다층 퍼셉트론에서 조정할 수 있는 하이퍼파라미터를 모두 나열해보기. 훈련 데이터에 다층 퍼셉트론이 과대적합되었다면 이를 해결하기 위해 하이퍼파라미터를 어떻게 조정해야 할까?

10. 심층 다층 퍼셉트론을 MNIST 데이터셋에 훈련해보기(keras.datasets.mnist.load_data() 함수를 사용해 데이터를 적재할 수 있음).
    98% 이상의 정확도를 얻을 수 있는지 확인.
    이 장에서 소개한 방법을 사용해 최적의 학습률을 찾아보기(즉 학습률을 지수적으로 증가시키면서 손실을 그래프로 그리고, 그 다음 손실이 다시 증가하는 지점을 찾음).
    모든 부가 기능을 추가해보기. 즉, 체크포인트를 저장하고, 조기 종료를 사용하고, 텐서보드를 사용해 학습 곡선 그리기