Week1

CH10 - 케라스를 사용한 인공 신경망 소개
    10.1 생물학적 뉴런에서 인공 뉴런까지
        10.1.1 생물학적 뉴런
        10.1.2 뉴런을 사용한 논리 연산
        10.1.3 퍼셉트론
        10.1.4 다층 퍼셉트론과 역전파
        10.1.5 회귀를 위한 다층 퍼셉트론
        10.1.6 분류를 위한 다층 퍼셉트론
    10.2 케라스로 다층 퍼셉트론 구현하기
        10.2.1 텐서플로 2 설치
        10.2.2 시퀀셜 API를 사용하여 이미지 분류기 만들기
        10.2.3 시퀀셜 API를 사용하여 회귀용 다층 퍼셉트론 만들기
        10.2.4 함수형 API를 사용해 복잡한 모델 만들기
        10.2.5 서브클래싱 API로 동적 모델 만들기
        10.2.6 모델 저장과 복원
        10.2.7 콜백 사용하기
        10.2.8 텐서보드를 사용해 시각화하기
    10.3 신경망 하이퍼파라미터 튜닝하기
        10.3.1 은닉층 개수
        10.3.2 은닉층의 뉴런 개수
        10.3.3 학습률, 배치 크기 그리고 다른 하이퍼파라미터
    10.4 연습문제

CH10 - 케라스를 사용한 인공 신경망 소개

인공 신경망(ANN) ~ 인공 신경망은 딥러닝의 핵심

다층 퍼셉트론(MLP)에 대해서도 알아보자

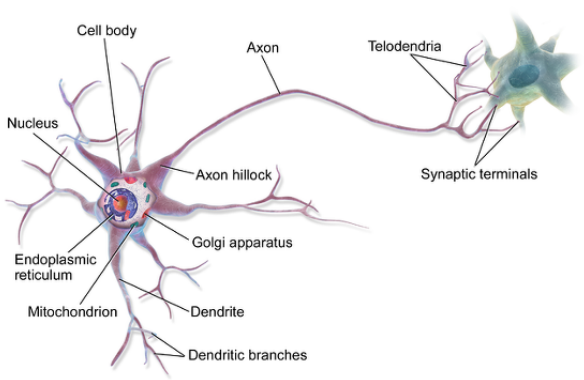
10.1 생물학적 뉴런에서 인공 뉴런까지

현재 인공 신경망의 재부흥

<근거>

- 방대한 데이터 양
- 하드웨어의 발전
- 훈련 알고리즘 향상
- 이론상 제한 → 실제로는 문제 X
- 투자와 진보의 선순환

10.1.1 생물학적 뉴런



10.1.2 뉴런을 사용한 논리 연산

매컬러와 피츠의 인공 뉴런 ~ 하나 이상의 이진 입력과 이진 출력 하나를 가짐

논리 명제도 계산 가능

10.1.3 퍼셉트론

가장 간단한 인공신경망 구조 ~ TLU(Threshold Logic Unit)이라는 인공 뉴런을 기반으로 함

입력 값: 이진값 아닌 어떤 숫자

입력 연결은 가중치와 연관 ~ 입력의 가중치 합을 계산한 뒤, 계산된 합에 step function을 적용하여 결과 출력

가중치 합 계산 → $z = w_1x_1 + w_2x_2 + ... + w_nx_n = x^Tw$

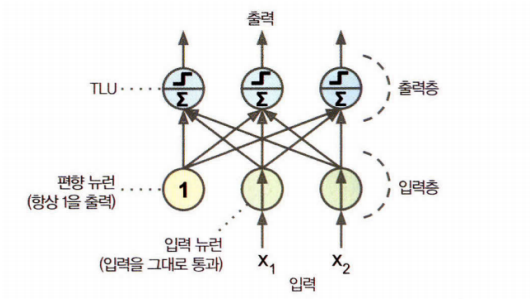
계단 함수 적용 → $h_w(x) = step(z)$

헤비사이드 계단 함수 ~ 가장 널리 사용되는 계단 함수

완전 연결층(밀집층): 한 층에 있는 모든 뉴런이 이전 층의 모든 뉴런과 연결되어 있을 때

입력 뉴런: 퍼셉트론의 입력 ~ 입력층: 모든 입력 뉴런

편향: 항상 1을 출력



완전 연결층 출력 계산 → $h_{W,b}(X) = \phi(XW + b)$

ϕ : 활성화 함수 activation function (인공 뉴런이 TLU일 경우 이 함수는 계단 함수)

헤브의 규칙(헤브 학습): 두 뉴런이 동시에 활성화될 때마다 이들 사리의 연결 가중치가 증가하는 경향이 있다

<퍼셉트론 학습 규칙>

- 네트워크가 예측할 때 만드는 오차 반영 → 변형된 규칙 사용하여 훈련
- 오차가 감소되도록 연결 강화
- 한번에 한개의 샘플 주입 → 각 샘플에 대한 예측 생성
- 잘못된 예측을 하는 출력 뉴런 → 올바른 예측을 할 수 있도록 연결된 가중치 강화
- $w_{i,j}^{(next\ step)} = w_{i,j} + \eta(y_i - \hat{y}_j)x_i$

→ 훈련 샘플이 선형적으로 구분될 수 있으며 이 알고리즘에 수렴한다

→ 퍼셉트론 수렴 이론

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

iris = load_iris()
x = iris.data[:,(2,3)]
y = (iris.target == 0).astype(np.int)

per_clf = Perceptron()
per_clf.fit(x,y)

y_pred = per_clf.predict([[2,0.5]])
```

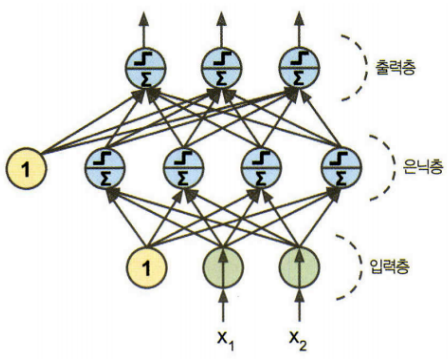
경사 하강법과 매우 유사

고정된 임계값을 기준으로 예측

퍼셉트론의 심각한 약점 → 일부 간단한 문제(XOR: 베타적 논리합 분류 문제) 해결 불가

⇒ 여러 개 쌓아 올리면 일부 제약 감소: **다층 퍼셉트론(MLP) → XOR 해결 가능!**

10.1.4 다층 퍼셉트론과 역전파



입력층과 가까운 층: 하위 층 // 출력층과 가까운 층: 상위 층

출력층을 제외하고 모든 층은,

- 편향 뉴런을 포함/
- 다음 층과 완전히 연결되어 있음

신호는 한 방향으로만 흐름 → **피드포워드 신경망(FNN)** 구조에 속함

은닉층을 여러개 쌓아 올린 인공 신경망: 심층 신경망(DNN)

다층 퍼셉트론을 훈련할 방법 → **역전파**(backpropagation) **훈련 알고리즘** 등장

= 경사 하강법: 그레디언트를 자동으로 계산 → **자동 미분**

<역전파 훈련 알고리즘>

훈련 세트 처리 반복 → 각 반복: 에포크(epoch)

1. 정방향 계산: 마지막 층인 출력층의 출력을 계산할 때까지 계속해서 계산
2. 손실 함수를 사용하여 네트워크의 출력 오차 측정
3. 출력 연결이 오차에 기여하는 정도 계산(연쇄 법칙 chain rule 적용)
4. 이전 층의 연결 가중치가 오차에 기여하는 정도 계산 → 입력층에 도달할 때까지 역방향 계산
5. 경사 하강법 수행 → 모든 연결 가중치 수정

✅ 은닉층의 연결 가중치를 랜덤하게 초기화하는 것이 중요!

다층 퍼셉트론의 중요한 변화!!

계단 함수 → **시그모이드(로지스틱)함수로 변경** → $\sigma(z) = \frac{1}{(1+exp(-z))}$

<활성화 함수>

- 시그모이드 함수
- 하이퍼볼릭 탄젠트 함수
- ReLU 함수

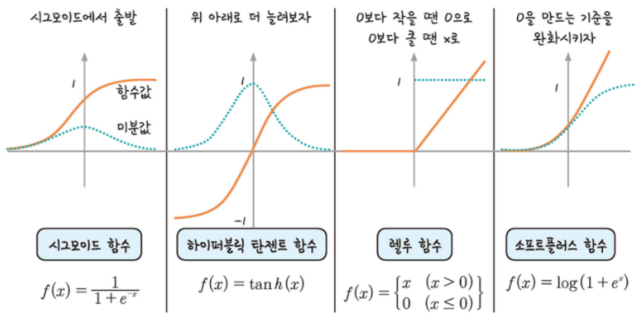


그림 9-4
여러 활성화 함수의 도입

Q. 활성화 함수는 왜 필요할까?

A. 선형 변환을 여러 개 연결해도 얻을 수 있는 것은 선형 변환 뿐
따라서, 층 사이 비선형성을 추가하지 않으면 아무리 많은 층을 쌓아도 하나의 층과 동일

10.1.5 회귀를 위한 다층 퍼셉트론

회귀 작업을 할 때 다층 퍼셉트론 사용

일반 회귀(하나의 값 예측), 출력: 예측된 값 → 하나의 출력 뉴런이 필요

다변량 회귀(동시에 여러 값 예측) → 출력 차원마다 출력 뉴런 필요

회귀용 다층 퍼셉트론을 만들 때, 일반적으로 활성화 함수를 사용 X

출력이 항상 양수이다! → 출력층에 ReLU 활성화 함수 사용 or softplus 함수 사용

❋ **softplus 함수:** $log(1 + exp(z))$

어떤 범위 안의 값을 예측하고 싶다! → 시그모이드 or 하이퍼볼릭 탄젠트 함수 사용

loss function: MSE // 이상치가 많으면 MAE, Huber(MSE와 MAE을 조합한 것)

하이퍼파라미터	일반적인 값
입력 뉴런 수	특성마다 하나
은닉층 수	문제에 따라 다름, 일반적으로 1에서 5 사이
은닉층의 뉴런 수	문제에 따라 다름, 일반적으로 10에서 100사이
출력 뉴런 수	예측 차원마다 하나
은닉층의 활성화 함수	ReLU(또는 SELU)
출력층의 활성화 함수	없음, 또는 (출력이 양수일 때) ReLU/softplus 나 (출력을 특정 범위로 제한할 때) logistic/tanh 사용
손실 함수	MSE나 (이상치가 있다면) MAE/Huber

회귀 MLP의 전형적인 구조

10.1.6 분류를 위한 다층 퍼셉트론

이진 분류 → 시그모이드 활성화 함수를 가진 하나의 출력 뉴런만 필요

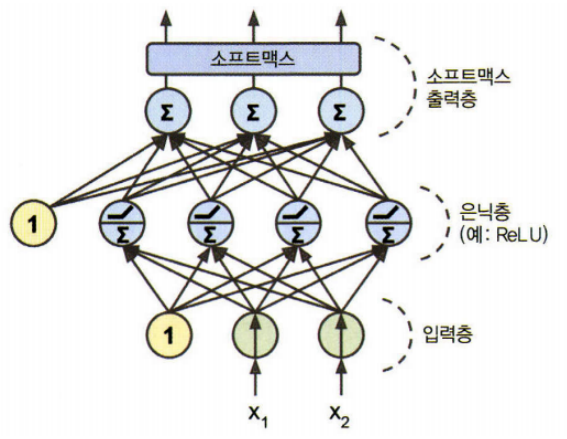
출력은 0과 1사이의 실수 → 양성 클래스에 대한 예측 확률로 해석

(음성 클래스에 대한 예측 확률 = 1- 양성 클래스에 대한 예측 확률)

다층 퍼셉트론: 다중 레이블 이진 분류 문제 처리에 용이(ex. 메일이 스팸인지 아닌지 + 긴급한 것인지 아닌지)

⇒ 긴급하지 않은 메일 / 긴급한 메일 / 긴급하지 않은 스팸 메일 / 긴급한 스팸 메일(오류로 추정)

- 클래스마다 하나의 출력 뉴런 필요
- 출력층에는 softmax 활성화 함수 사용(모든 예측 확률을 0과 1사이로 만들고 더했을 때 1이 됨)
- 이를 다중 분류라고 부름



loss function: 크로스 엔트로피 손실(=로그 손실) 사용 → 확률 분포 예측

하이퍼파라미터	이진 분류	다중 레이블 분류	다중 분류
입력층과 은닉층	회귀와 동일	회귀와 동일	회귀와 동일
출력 뉴런 수	1개	레이블마다 1개	클래스마다 1개
출력층의 활성화 함수	로지스틱 함수	로지스틱 함수	소프트맥스 함수
손실 함수	크로스 엔트로피	크로스 엔트로피	크로스 엔트로피

분류 MLP의 전형적인 구조

10.2 케라스로 다층 퍼셉트론 구현하기

10.2.1 텐서플로 2 설치

코랩 사용

10.2.2 시퀀셜 API를 사용하여 이미지 분류기 만들기

Fashion MNIST 데이터 사용

- 10개의 클래스
- 28 * 28 픽셀 크기의 흑백 이미지 70,000개
- 형태가 정확히 같지만 손글씨가 아니라 패션 아이템을 나타내는 이미지

<케라스를 이용하여 데이터 셋 적재하기>

```
fashion_mnist = keras.datasets.fashion_mnist
(x_train_full, y_train_full), (x_test, y_test) = fashion_mnist.load_data()
```

사이킷런을 사용해 적재하였을 때와 차이가 있음

Aa 이름	사이킷런	케라스
크기와 배열	784 크기/ 1D배열	28 * 28 크기의 배열
픽셀 강도	실수(0.0 ~ 255.0)	정수(0 ~ 255)

```
#훈련 세트의 크기와 데이터 타입 확인
>>> x_train_full.shape
(60000, 28, 28)
>>> x_train_full.dtype
dtype('uint8')

#검증 세트 만들기
#입력 특성의 스케일 조정
x_valid, x_train = x_train_full[:5000]/255.0, x_train_full[5000:]/255.0
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
x_test = x_test/255.0

#클래스 이름 리스트 생성
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal',
               'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

#확인
>>> class_names[y_train[0]]
'Coat'
```

<시퀀셜 API를 사용하여 모델 만들기>

```
#신경망 생성
#두 개의 은닉층으로 이루어진 분류용 다층 퍼셉트론
model = keras.models.Sequential() #모델 생성
model.add(keras.layers.Flatten(input_shape=[28,28])) #입력층
model.add(keras.layers.Dense(300, activation="relu")) #은닉층
model.add(keras.layers.Dense(100, activation="relu")) #은닉층
model.add(keras.layers.Dense(10, activation="softmax")) #출력층

>>> model.summary() #첫 번째 은닉층은 파라미터의 갯수를 확인했을 때, 과대적합의 위험 있음을 알 수 있다.
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
_________________________________________________________________

>>> model.layers
[<keras.layers.core.Flatten at 0x7f2954c852d0>,
 <keras.layers.core.Dense at 0x7f2954e2b7d0>,
 <keras.layers.core.Dense at 0x7f2959257650>,
 <keras.layers.core.Dense at 0x7f2954c87650>]

>>>hidden1 = model.layers[1]
>>> hidden1.name
'dense'

>>>model.get_layer('dense') is hidden1
True

#파라미터로의 접근
>>> weights, biases = hidden1.get_weights()
>>> weights
array([[-0.01202889, -0.02520375, -0.02020528, ..., -0.05904793,
         0.0494606 , -0.06317092],
       [-0.05904678,  0.05315712, -0.06610563, ...,  0.06989942,
         0.02853368, -0.00395272],
       [ 0.04747478, -0.06598961,  0.00152043, ...,  0.03865387,
         0.03502854, -0.01272675],
       ...,
       [ 0.029612  , -0.01239874, -0.01016549, ...,  0.02901292,
         0.00269286,  0.02409095],
       [-0.03735555, -0.06553811, -0.04395407, ..., -0.01992206,
        -0.03858028, -0.03149897],
       [-0.01325106,  0.03015194,  0.06456205, ...,  0.03242714,
        -0.05206587,  0.03710809]], dtype=float32)

>>> weights.shape
(784, 300)
>>> biases
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       ... , 0., 0., 0., 0.], dtype=float32)
>>> biases.shape
(300,)
```

Dense 층은 연결 가중치를 무작위로 초기화, 편향은 0으로 초기화 → for. 대칭성 붕괴

<모델 컴파일>

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

'sparse_categorical_crossentropy' 손실 사용

- 레이블이 정수 하나(샘플마다 타깃 클래스 인덱스 하나 존재)
- 클래스가 배타적

☑️ 샘플마다 클래스별 타깃 확률을 가지고 있다면, 'categorical_crossentropy' 손실 사용

옵티마이저 'sgd'

- 기본 확률적 경사 하강법을 사용하여 모델을 훈련한다 라는 의미
- = 역전파 알고리즘 수행
- 학습률 튜닝이 중요(디폴트: lr = 0.01)

정확도 측정

- accuracy

<모델 훈련과 평가>

```
>>> history = model.fit(x_train, y_train, epochs=30, validation_data=(x_valid, y_valid))
Epoch 1/30
1719/1719 [==============================] - 8s 3ms/step - loss: 0.7260 - accuracy: 0.7606 - val_loss: 0.5170 - val_accuracy: 0.8256
Epoch 2/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.4950 - accuracy: 0.8278 - val_loss: 0.4490 - val_accuracy: 0.8506
.
.
.
Epoch 29/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2309 - accuracy: 0.9166 - val_loss: 0.2953 - val_accuracy: 0.8920
Epoch 30/30
1719/1719 [==============================] - 6s 3ms/step - loss: 0.2267 - accuracy: 0.9197 - val_loss: 0.3034 - val_accuracy: 0.8908
```

훈련 세트의 성능 >>>> 검증 세트 성능 → 과대적합 or 버그 가능성 있음

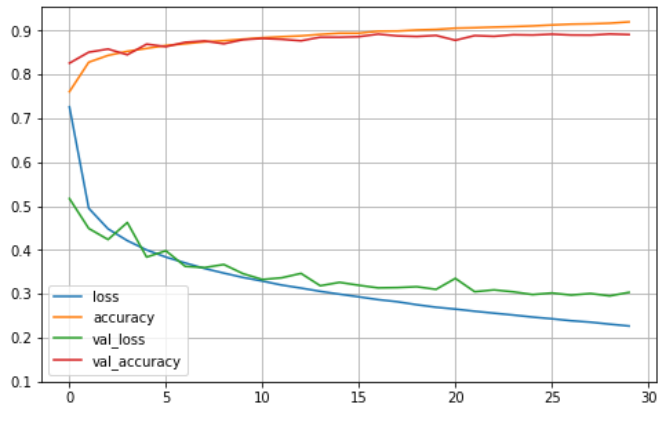
에포크가 지날 수록 훈련 손실 감소

훈련 세트 편중 → 클래스별 가중치 부여: class_weight 매개변수 지정

샘플별 가중치 부여: sample_weight 매개변수 지정

#둘 다 지정됐을 경우, 두 값을 곱하여 사용

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

pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
plt.gca().set_ylim(0.1) #수직축의 범위를 0과1 사이로 설정
plt.show();
```



학습 곡선(learning curve)

- 각 정확도 꾸준히 상승
- 각 손실은 모두 감소
- 훈련 곡선과 검증 곡선이 가까움 → 과대적합 X
- 훈련 손실: 에포크가 진행되는 동안 계산 // 검증 손실: 에포크가 끝난 후 계산
  → 훈련 곡선은 에포크의 절반 만큼 왼쪽으로 이동
  → 두 곡선 거의 일치 !

모델이 만족스럽지 않다면, 처음으로 돌아가 **하이퍼파라미터 튜닝**

1. 학습률 확인
2. 다른 옵티마이저
3. 층 개수, 층에 있는 뉴런 개수, 은닉층이 사용하는 활성화 함수 등
4. 배치 크기(fit( ) 메서드 → default: batch_size = 32)

모델이 만족스럽다면, 배포 전 모델 평가(일반화 오차 추정)

```
#evaluate( ) 메서드 사용
>>> model.evaluate(x_test, y_test)
313/313 [==============================] - 1s 3ms/step - loss: 0.3379 - accuracy: 0.8824
[0.337948739528656, 0.8823999762535095]
```

- 일반적으로 검증 세트보다 테스트 세트에서의 성능이 낮음
- 하이퍼파라미터 튜닝 → 검증 세트에서 이루어졌기 때문

<모델을 사용해 예측을 만들기>

```
#predict()메서드 사용
>>> x_new = x_test[:3]
>>> y_proba = model.predict(x_new)
>>> y_proba.round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.02, 0.  , 0.03, 0.  , 0.95],
       [0.  , 0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)

#가장 높은 확률을 가진 클래스
>>> import numpy as np
>>> y_pred = np.argmax(y_proba, axis=1) #교재에서 사용한 predict_classes는 21년 1월 1일 후로 삭제됨.
>>> y_pred
array([9, 2, 1])

>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')

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

10.2.3 시퀀셜 API를 사용하여 회귀용 다층 퍼셉트론 만들기

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
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)
x_train, x_valid, y_train, y_valid = train_test_split(x_train_full, y_train_full)

scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_valid = scaler.transform(x_valid)
x_test = scaler.transform(x_test)
```

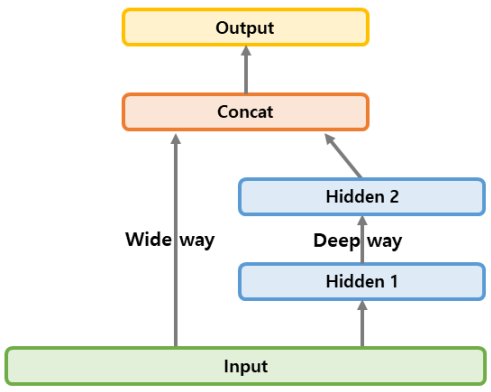
- 분류에서 했던 방법과 비슷

- 출력층이 활성화 함수가 없는 하나의 뉴런을 가짐(✅ 하나의 값 예측)
- 손실 함수: 평균 제곱 오차 사용
- 사용한 데이터는 잡음 多 → 과대적합을 막고자 뉴런 수가 적은 은닉층 하나만 사용

```
>>> model = keras.models.Sequential([keras.layers.Dense(30, activation="relu", input_shape = x_train.shape[1:]),
                                     keras.layers.Dense(1)])

>>> model.compile(loss='mean_squared_error', optimizer='sgd')

>>> history = model.fit(x_train, y_train, epochs=20, validation_data=(x_valid, y_valid))
>>> mse_test = model.evaluate(x_test, y_test)

>>> x_new = x_test[:3]
>>> y_pred = model.predict(x_new)

Epoch 1/20
363/363 [==============================] - 2s 4ms/step - loss: 1.0201 - val_loss: 0.5163
Epoch 2/20
363/363 [==============================] - 1s 4ms/step - loss: 0.4473 - val_loss: 2.5473
.
.
.
Epoch 19/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3458 - val_loss: 0.4143
Epoch 20/20
363/363 [==============================] - 1s 3ms/step - loss: 0.3477 - val_loss: 0.3647
162/162 [==============================] - 0s 2ms/step - loss: 0.3581
```

10.2.4 함수형 API를 사용해 복잡한 모델 만들기

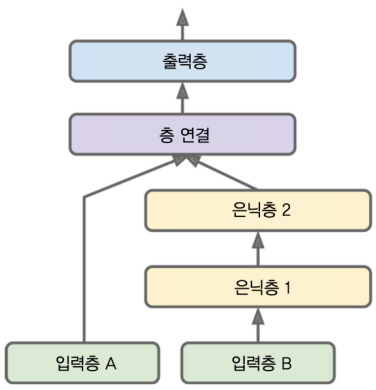
순차적이지 않은 신경망: **와이드 & 딥 신경망**



Wide & Deep Learning

- 입력의 일부 또는 전체가 출력층에 바로 연결
- 복잡한 패턴(깊게 쌓은 층을 사용한) & 간단한 규칙(짧은 경로를 사용한)을 모두 학습할 수 있음

```
input_ = keras.layers.Input(shape=x_train.shape[1:]) #입력층
hidden1 = keras.layers.Dense(30, activation='relu')(input_) #은닉층 1
hidden2 = keras.layers.Dense(30, activation='relu')(hidden1) #은닉층 2
concat = keras.layers.Concatenate()([input_, hidden2]) #층 연결
output = keras.layers.Dense(1)(concat) #출력층
model = keras.Model(inputs = [input_], outputs = [output]) #모델 생성
```




```
#일부 특성은 짧은 경로로, 다른 특성은 깊은 경로로
#여러 입력을 사용
input_A = keras.layers.Input(shape=[5], name='wide_input')
input_B = keras.layers.Input(shape=[6], name='deep_input')
hidden1 = keras.layers.Dense(30, activation='relu')(input_B)
hidden2 = keras.layers.Dense(30, activation='relu')(hidden1)
concat = keras.layers.Concatenate()([input_A, hidden2])
output = keras.layers.Dense(1, name='output')(concat)
model = keras.Model(inputs = [input_A, input_B], outputs = [output])
```

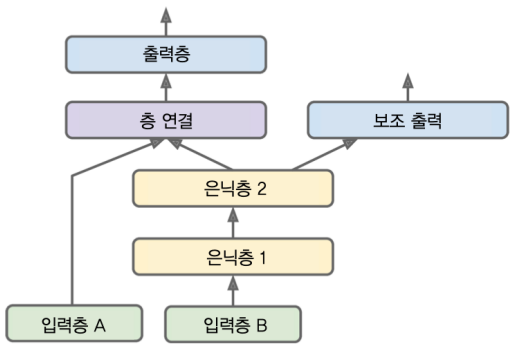
- fit( )을 호출할 때 x_train만 전달하는 것이 아님 → (x_train_A, x_train_B)로 전달
- evaluate( ), predict( )도 마찬가지임

```
>>> model.compile(loss='mse', optimizer=keras.optimizers.SGD(learning_rate=1e-3))

>>> x_train_A, x_train_B = x_train[:,:5], x_train[:,2:]
>>> x_valid_A, x_valid_B = x_valid[:,:5], x_valid[:,2:]
>>> x_test_A, x_test_B = x_test[:,:5], x_test[:,2:]
>>> x_new_A, x_new_B = x_test_A[:3], x_test_B[:3]
```

```
>>> history = model.fit((x_train_A, x_train_B), y_train, epochs=20,
                        validation_data=((x_valid_A, x_valid_B), y_valid))
>>> mse_test = model.evaluate((x_test_A, x_test_B), y_test)
>>> y_pred = model.predict((x_new_A, x_new_B))

Epoch 1/20
363/363 [==============================] - 2s 3ms/step - loss: 2.1259 - val_loss: 0.9569
Epoch 2/20
363/363 [==============================] - 1s 3ms/step - loss: 0.8205 - val_loss: 0.7564
Epoch 3/20
363/363 [==============================] - 1s 3ms/step - loss: 0.7030 - val_loss: 0.6858
.
.
.
Epoch 19/20
363/363 [==============================] - 1s 3ms/step - loss: 0.4768 - val_loss: 0.4674
Epoch 20/20
363/363 [==============================] - 1s 4ms/step - loss: 0.4739 - val_loss: 0.4675
162/162 [==============================] - 0s 2ms/step - loss: 0.4774
```

여러 개의 출력이 필요한 경우

- 여러 출력이 필요한 작업(ex. 회귀 작업과 분류 작업을 함께하는 경우)
- 동일한 데이터에서 독립적인 여러 작업을 수행할 때
  (ex. 다중분류작업 → 사람의 표정 분류 후 안경 착용 유무 분류)
- 규제 기법으로 사용




```
#출력층까지는 이전과 동일
input_A = keras.layers.Input(shape=[5], name='wide_input')
input_B = keras.layers.Input(shape=[6], name='deep_input')
hidden1 = keras.layers.Dense(30, activation='relu')(input_B)
hidden2 = keras.layers.Dense(30, activation='relu')(hidden1)
concat = keras.layers.Concatenate()([input_A, hidden2])

output = keras.layers.Dense(1, name='main_output')(concat)
aux_output = keras.layers.Dense(1, name='aux_output')(hidden2)
model = keras.Model(inputs = [input_A, input_B], outputs = [output, aux_output])
```

각 출력은 자신만의 손실 함수 필요 → 컴파일할 때 소실의 리스트 전달

나열된 손실의 합 ⇒ 최종 손실

주 출력의 손실에 더 많은 가중치 부여

```
model.compile(loss=['mse', 'mse'], loss_weights=[0.9, 0.1], optimizer='sgd')
```

각 출력에 대한 레이블 제공

```
>>> history = model.fit([x_train_A, x_train_B], [y_train, y_train],
                        epochs=20, validation_data=([x_valid_A, x_valid_B], [y_valid, y_valid]))
Epoch 1/20
363/363 [==============================] - 3s 6ms/step - loss: 0.9540 - main_output_loss: 0.8599 - aux_output_loss: 1.8010 - val_loss: 0.6034 - val_main_output_loss: 0.5496 - val_aux_output_loss: 1.0883
Epoch 2/20
363/363 [==============================] - 2s 5ms/step - loss: 0.5891 - main_output_loss: 0.5367 - aux_output_loss: 1.0601 - val_loss: 0.4754 - val_main_output_loss: 0.4309 - val_aux_output_loss: 0.8756
Epoch 3/20
.
.
.
Epoch 19/20
363/363 [==============================] - 2s 5ms/step - loss: 0.4053 - main_output_loss: 0.3933 - aux_output_loss: 0.5138 - val_loss: 0.3423 - val_main_output_loss: 0.3282 - val_aux_output_loss: 0.4690
Epoch 20/20
363/363 [==============================] - 2s 5ms/step - loss: 0.3823 - main_output_loss: 0.3682 - aux_output_loss: 0.5087 - val_loss: 0.3476 - val_main_output_loss: 0.3354 - val_aux_output_loss: 0.4568
```

모델 평가 → 개별 손실과 총 손실 반환

```
>>> total_loss, main_loss, aux_loss = model.evaluate(
        [x_test_A, x_test_B], [y_test, y_test])

162/162 [==============================] - 0s 3ms/step - loss: 0.4059 - main_output_loss: 0.3958 - aux_output_loss: 0.4973
```

각 출력에 대한 예측 반환

```
y_pred_main, y_pred_aux = model.predict([x_new_A, x_new_B])
```

10.2.5 서브클래싱 API로 동적 모델 만들기

시퀀셜 API, 함수형API: 선언적

사용할 층과 연결 방식 먼저 정의 → 모델에 데이터 주입 → 훈련 및 추론

- 모델의 저장, 복사, 공유가 쉬움

- 모델의 구조를 출력, 분석하기 좋음
- 에러를 일찍 발견할 수 있음
- 디버깅도 쉬움

동적인 구조가 필요할 때는 정적이라는 것이 단점

⇒ 명령형 프로그래밍 스타일 - 서브클래싱 API !!

```
class WideAndDeepModel(keras.Model):
    def __init__(self, units=30, activation='relu', **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = keras.layers.Dense(units, activation=activation)
        self.hidden2 = keras.layers.Dense(units, activation=activation)
        self.main_output = keras.layers.Dense(1)
        self.aux_output = keras.layers.Dense(1)

    def call(self, inputs):
        input_A, input_B = inputs
        hidden1 = self.hidden1(input_B)
        hidden2 = self.hidden2(hidden1)
        concat = keras.layers.concatenate([input_A, hidden2])
        main_output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return main_output, aux_output

model = WideAndDeepModel()
```

→ 높은 유연성을 필요로 하지 않는다면 시퀀셜 API과 함수형API를 사용하는 것이 좋음

10.2.6 모델 저장과 복원

```
#모델 저장
model = keras.models.Sequential([...])
model.compile([...])
model.fit([...])
model.save('my_keras_model.h5')

#모델 불러오기
model = keras.models.load_model('my_keras_model.h5')
```

10.2.7 콜백 사용하기

대규모의 데이터 셋 학습 시 정보를 잃지 않으려면, 훈련 도중 일정 간격으로 체크포인트 저장

기본적으로 매 에포크 의 끝에서 호출

```
[...] #모델을 만들고 컴파일 하기
checkpoint_cb = keras.callbacks.ModelCheckpoint('my_keras_model.h5')
history = model.fit(x_train, y_train, epochs=10, callbacks=[checkpoint_cb])

#조기 종료 구현1
#최상의 검증 세트 점수에서만 모델 저장
checkpoint_cb = keras.callbacks.ModelCheckpoint('my_keras_model.h5',
                                                save_best_only = True)
history = model.fit(x_train, y_train, epochs=10,
                    validation_data=(x_valid, y_valid),
                    callbacks=[checkpoint_cb])
model = keras.models.load_model('my_keras_model.h5') #최상의 모델로 복원

#조기 종료 구현2
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
                                                  restore_best_weights=True)
history = model.fit(x_train, y_train, epochs=10, #모델이 향상되지 않으면 자동으로 훈련이 멈추기 때문에 크게 지정해도 됨
                    validation_data=(x_valid, y_valid),
                    callbacks=[checkpoint_cb])
model = keras.models.load_model('my_keras_model.h5')

#더 많은 제어 - 사용자 정의 콜백
class PrintValTrainRatioCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        print('\nval/train: {:.2f}'.format(logs['val_loss'] / logs['loss']))
```

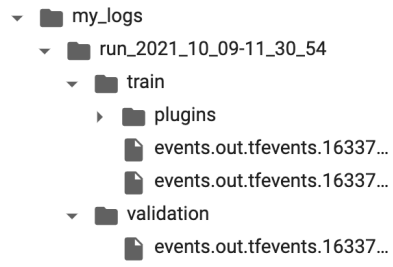
10.2.8 텐서보드를 사용해 시각화하기

이벤트 파일(이진 로그 파일) - 시각화하려는 데이터 출력

각각의 이진 데이터 레코드: 서머리(summary)

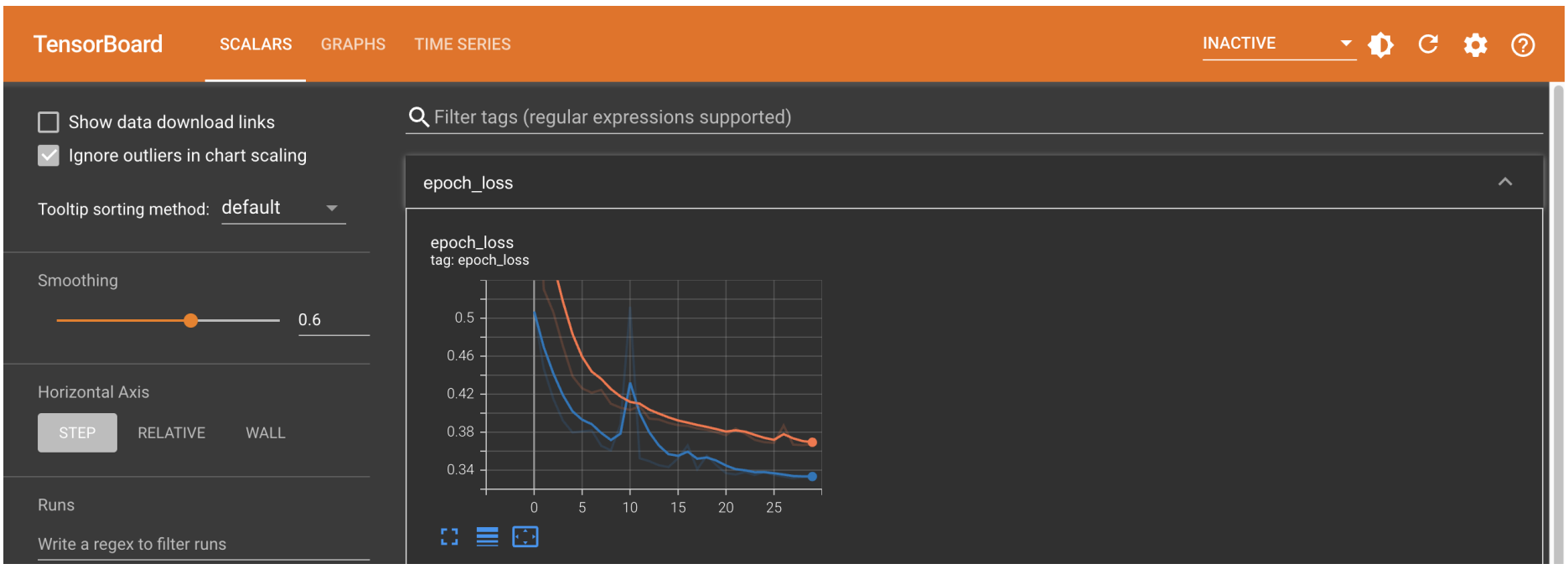
```
#사용할 루트 로그 디렉토리 정의
import os
root_logdir = os.path.join(os.curdir, 'my_logs')

#실행할 때마다 다른 서브디렉토리 경로 생성
def get_run_logdir():
    import time
    run_id = time.strftime('run_%Y_%m_%d-%H_%M_%S')
    return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir()
```

```
#모델 구성과 컴파일
model = keras.models.Sequential([keras.layers.Dense(30, activation="relu",
                                                    input_shape = x_train.shape[1:]),keras.layers.Dense(1)])

model.compile(loss='mean_squared_error', optimizer='sgd')

tensorboard_cb = keras.callbacks.TensorBoard(run_logdir) #텐서보드 콜백이 로그 디렉토리 생성
history = model.fit(x_train, y_train, epochs=30,
                    validation_data=(x_valid, y_valid),
                    callbacks=[tensorboard_cb])
```



실행마다 하나의 디렉토리 생성

```
#텐서보드 서버 시작
%load_ext tensorboard
%tensorboard --logdir=./my_logs --port=6006
```



10.3 신경망 하이퍼파라미터 튜닝하기

유연성 - 최적의 하이퍼파라미터 찾기

- 많은 조합을 시도해보고 어떤 것이 검증 세트에서 가장 좋은 점수를 내는지 확인

```
#케라스 모델을 사이킷런 추정기처럼 보이도록
#케라스 모델 생성 및 컴파일 함수 생성
def bulid_model(n_hidden=1, n_neurons=30, learning_rate=3e-3, input_shape=[8]):
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=input_shape))
    for layer in range(n_hidden):
        model.add(keras.layers.Dense(n_neurons, activation='relu'))
    model.add(keras.layers.Dense(1))
    optimizer = keras.optimizers.SGD(learning_rate=learning_rate)
    model.compile(loss='mse', optimizer=optimizer)
    return model

#케라스 모델을 감싸는 레퍼 생성
keras_reg = keras.wrappers.scikit_learn.KerasRegressor(bulid_model)

#케라스 모델을 일반 사이킷런 회귀 추정기처럼 사용 가능
keras_reg.fit(x_train, y_train, epochs=100,
              validation_data=(x_valid, y_valid),
              callbacks=[keras.callbacks.EarlyStopping(patience=10)])
mse_test = keras_reg.score(x_test, y_test)
y_pred = keras_reg.predict(x_new)

#랜덤 탐색 (1시간 걸림...)
from scipy.stats import reciprocal
from sklearn.model_selection import RandomizedSearchCV

params_distribs = {
    'n_hidden': [0,1,2,3],
    'n_neurons': np.arange(1,100),
    'learning_rate': reciprocal(3e-4, 3e-2)
}

rnd_search_cv = RandomizedSearchCV(keras_reg, params_distribs, n_iter=10, cv=3)
rnd_search_cv.fit(x_train, y_train, epochs=100,
                  validation_data=(x_valid, y_valid),
                  callbacks=[keras.callbacks.EarlyStopping(patience=10)])

#최상의 하이퍼파라미터와 훈련된 케라스 모델
>>> rnd_search_cv.best_params_
{'learning_rate': 0.02626563927521171, 'n_hidden': 2, 'n_neurons': 91}

>>> rnd_search_cv.best_score_
-0.2984093427658081

>>> model = rnd_search_cv.best_estimator_.model #모델 생성
```

- 랜덤 탐색을 이용하는 것은 어렵지 않지만 데이터 셋이 커졌을 때 효율적인 방법은 아님~~(공간의 제약)~~
- 수동으로 보조하여 문제 완화 가능(범위를 점점 줄여서 탐색하는 방식)
  → 이것도 좋은 방식은 아님
- **효율적인 공간 탐색 기법**
  - 탐색 지역이 좋다고 판명될 때, 더 탐색을 수행하는 것
  - Hyperpot
  - Hyperas, kopt, Talos
  - Keras Tuner
  - Scikit - Optimize(skopt)
  - Spearmint

- Hyperband
- Sklearn - Deap

10.3.1 은닉층 개수

뉴런 수만 무한하다면 은닉층 하나로 어떤 함수도 근사할 수 있음

하지만, 복잡한 문제에서는 심층 신경망의 파라미터 효율성이 훨씬 좋음

계층 구조 → 입력층 > 아래쪽 은닉층 > 중간 은닉층 > 위쪽 은닉층 >출력층
저수준의 구조 모델링
저수준 구조 연결 중간 수준 구조 모델링
중간 수준 구조 연결 고수준 구조 모델링

- 학습 시간 단축
- 새로운 데이터에 일반화되는 능력 향상
  새로운 신경망에서 처음 몇 개 층의 가중치와 편향을 첫 번째 신경망의 층에 있는 가중치와 편향값으로 초기화
  저수준 구조를 학습할 필요 X → 고수준 구조만 학습!!
  ⇒ **전이학습**

10.3.2 은닉층의 뉴런 개수

과거에는 깔때기 구성 (은닉층마다 300개 → 200개 → 100개)

현재는 모든 은닉층에 같은 크기 사용

튜닝할 하이퍼파라미터 → 층 마다 한 개씩 X // 전체 통틀어 한 개

⇒ 데이터 셋에 따라 다르지만 첫 번째 은닉층을 크게하는 것이 도움 됨

과대적합을 사전에 방지 X

필요한 것보다 더 많은 층과 뉴런을 가진 모델 선택 → 조기 종료 or 규제 기법 사용

⇒ '스트레치 팬츠' 방식

- 병목층을 피할 수 있음
- 유용한 정보를 유지할 확률이 높음
- 뉴런 2개 - 2D 데이터만 출력 가능 → 3D 데이터 처리 : 일부 정보 손실

🔥 **일반적으로 층의 뉴런 수보다 층 수를 늘리는 것이 유리**

10.3.3 학습률, 배치 크기 그리고 다른 하이퍼파라미터

<학습률>

가장 중요한 하이퍼파라미터

최적의 학습률 = 최대 학습률의 절반 정도

- 낮은 학습률 - - - 매우 큰 학습률까지 반복 훈련 (반복마다 일정한 값을 학습률에 곱하기)
- 손실이 다시 상승하는 지점 有 → 이 지점에서 조금 아래 = 최적의 학습률
  상승점보다 약 10배 낮은 지점

<옵티마이저>

11장에서

...

<배치 크기>

모델 성능과 훈련 시간에 큰 영향

큰 배치 크기 → 하드웨어 가속기(ex. GPU)를 효율적으로 활용
초당 더 많은 샘플 처리 가능 !

- GPU 램에 맞는 가장 큰 배치 크기 사용 권장
- 실전에서는 훈련 초기 불안정한 훈련 주의 - -
- 2 ~ 32의 미니 배치를 사용하는 것이 바람직하다(너무 큰 배치 크기는 일반화 성능에 영향을 미친다)

↑↓

- 아니다! 다양한 기법으로 매우 큰 배치 크기(8,192까지)도 사용할 수 있다!

⇒ 전략) 학습률 예열을 사용해 큰 배치 크기를 시도 - ~~불안정적이면, → 작은 배치 크기 사용~~

<활성화 함수>

일반적으로 RELU가 모든 은닉층에 좋은 기본값

<반복 횟수>

대부분 이 값은 튜닝이 필요 없음

대신 조기 종료 사용

10.4 연습문제

연습문제 풀이