# 10장 케라스를 사용한 인공신경망 소개

## 감사의 글

자료를 공개한 저자 오렐리앙 제롱에게 깊은 감사를 드립니다. 이와 더불어 한빛미디어로부터 강의준비에 필요한 자료를 지원받았음을 밝히며, 이에 대해 진심어린 감사를 전합니다.

- 인공신경망
    - 딥러닝의 핵심
    - 강력하며 확장성이 좋음

- 인공신경망의 응용: 대규모 머신러닝 문제 다루기에 적합
    - 구글 이미지: 수백만 개의 이미지 분류
    - 애플의 시리: 음성인식 서비스
    - 유튜브: 가장 좋은 비디오 추천
    - 딥마인드의 알파고: 스스로 기보를 익히면서 학습하는 바둑 프로그램

- 퍼셉트론
    - 가장 단순한 인공신경망 구조 중 하나
    - 프랑크 로젠블라트가 1957년에 발표.

## 주요 내용

- 퍼셉트론 소개
- 다층 퍼셉트론과 역전파
- 케라스(keras)를 이용한 다층 퍼셉트론 구현
- 텐서보드를 활용한 시각화
- 신경망 하이퍼파라미터 튜닝

# 퍼셉트론 소개

- TLU(threshold logic unit) 또는 LTU(linear threshold unit) 라 불리는 인공 뉴런 활용
- 입력/출력: 숫자
- 모든 입력은 가중치와 연결됨.

TLU

- 입력값과 가중치를 곱한 값들의 합에 계단함수(step function) 적용.

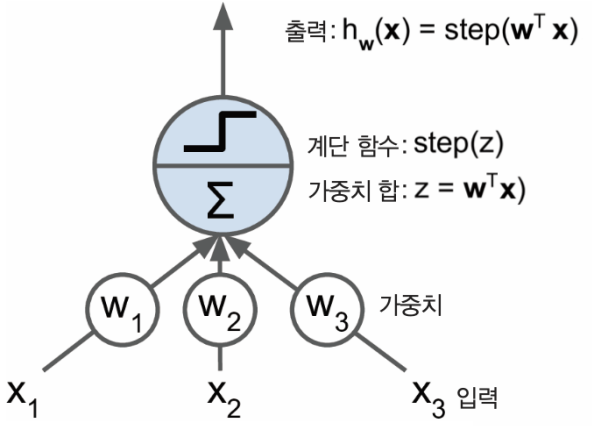
$$
\begin{aligned}
z &= \mathbf{x}^T \mathbf{w} \\
&= x_1 w_1 + x_2 w_2 + \cdots + x_n w_n
\end{aligned}
$$

$$
h_{\mathbf{w}}(\mathbf{x}) = step(z)
$$

- $n$: 특성 수

## 예제

- 3개의 특성을 갖는 샘플 하나가 입력되면 가중치와 곱한 후 합을 계산
- 이후 계단함수를 통과시킨 결과를 출력
- 보통은 편향에 대한 가중치고 함께 고려. (잠시 뒤에 설명)



## 계단함수

- 가장 많이 사용되는 계단함수: Heaviside 계단함수와 sign 함수

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

## TLU와 선형 이진분류

- 하나의 TLU를 간단한 이진분류기로 활용 가능
- 모든 입력값(특성)의 선형 조합을 계산한 후에 임곗값을 기준으로 양성/음성 분류
- 작동은 로지스틱 회귀 또는 선형 SVM 분류기와 비슷.
- TLU 모델 학습 = 최적의 가중치 $w_i$를 찾기

## 퍼셉트론 정의

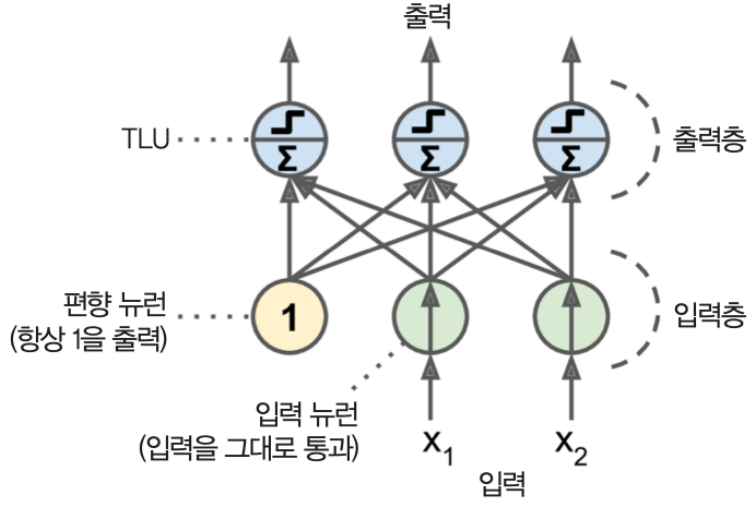
- 하나의 층에 여러 개의 TLU로 구성됨.
- TLU 각각은 모든 입력과 연결됨.

**입력층**

- 입력 뉴런으로 구성된 층
- 입력 뉴런: 입력을 받아 그대로 통과시켜 출력하는 뉴런
- 편향값 1을 항상 출력하는 편향 뉴런과 여러 개의 입력 뉴런이 함께 사용됨.

## 예제

- 입력 두 개와 출력 세 개로 구성된 퍼셉트론



## 퍼셉트론 학습 알고리즘

- 오차가 감소되도록 가중치를 조절하며 뉴런 사이의 관계를 강화시킴.

- 하나의 샘플이 입력될 때 마다 예측한 후에 오차를 계산하여 오차가 줄어드는 방향으로 가중치 조절.

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

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

- $w_{i,j}$: 가중치 행렬 $W$의 $i$행 $j$열의 값.
- $x_i$: 입력 샘플의 $i$번째 속성
- $\hat{y}_j$: $j$번째 출력값
- $y_j$: $j$번째 출력값에 대한 타깃
- $\eta$: 학습률

## 퍼셉트론과 선형성

- 각 출력 뉴런의 결정경계가 선형
- 따라서 퍼셉트론도 복잡한 패턴 학습 못함. 하지만 ...
- 퍼셉트론 수렴 이론: 선형적으로 구분될 수 있는 모델은 언제나 학습 가능

## 사이킷런의 퍼셉트론 모델

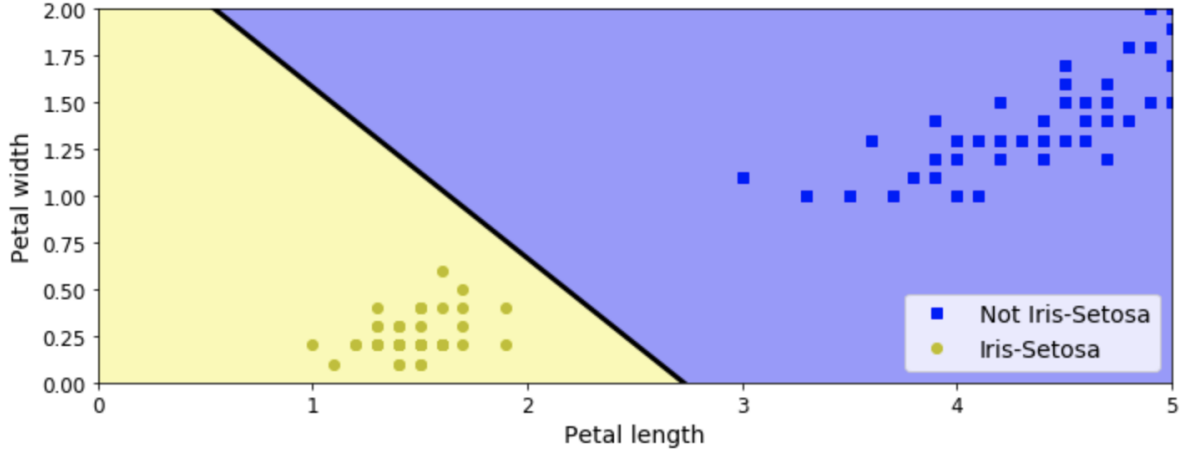
- Perceptron 클래스 활용

## 예제: 붓꽃 데이터셋 분류

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

iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(np.int)

per_clf = Perceptron(max_iter=1000, tol=1e-3, random_state=42)
per_clf.fit(X, y)
```


2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Petal width
0
1
2
3
4
5
Petal length
Not Iris-Setosa
Iris-Setosa

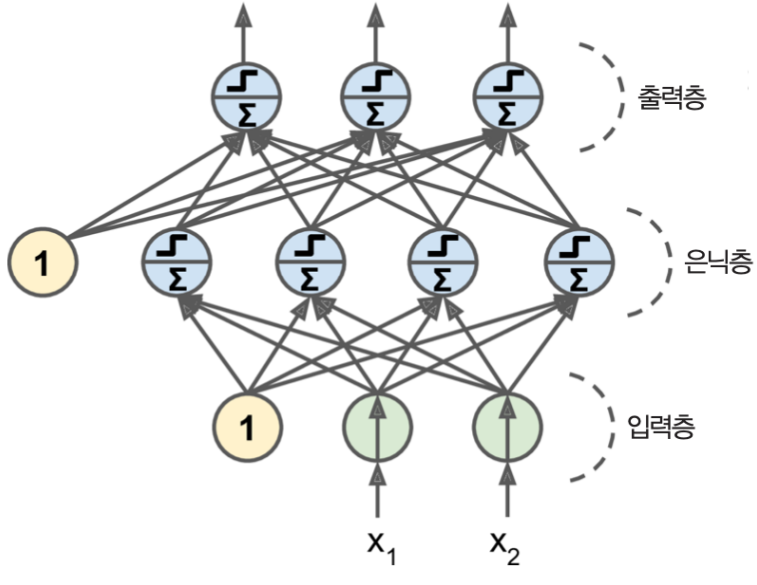
## 퍼셉트론의 특징

- 아래 옵션을 사용할 경우 SGDClassifier 와 동일하게 작동
    - `loss="perceptron"`
    - `learning_rate="constant"`
    - `eta0=1, penalty=None`
- 클래스 확률 지원 없음. 따라서 로지스틱 회귀가 보다 선호됨.
- 퍼셉트론은 매우 단순한 경우만 해결 가능.
- 하지만 퍼셉트론을 여러 개 쌓아올리면 꽤 강력한 인공신경망 구성함.

# 다층 퍼셉트론(MLP)과 역전파

- 다층 퍼셉트론(multilayer perceptron, MLP): 퍼셉트론을 여러 개 쌓아올린 인공신경망

- 구성은 다음과 같음.
    - 한 개의 입력층
    - 여러 개의 은닉층
    - 한 개의 출력층



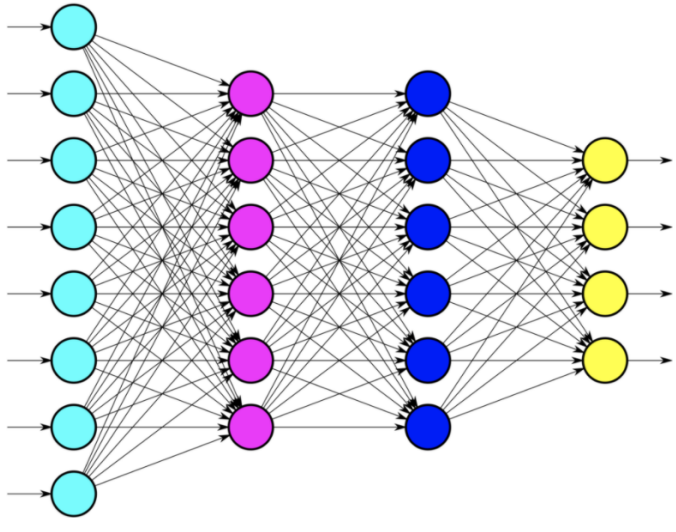
## 완전연결 층(밀집 층)

- 층에 속한 각각의 뉴런이 이전 층의 모든 뉴런과 연결되어 있을 때를 가리킴.

- 예제: 퍼셉트론의 출력층

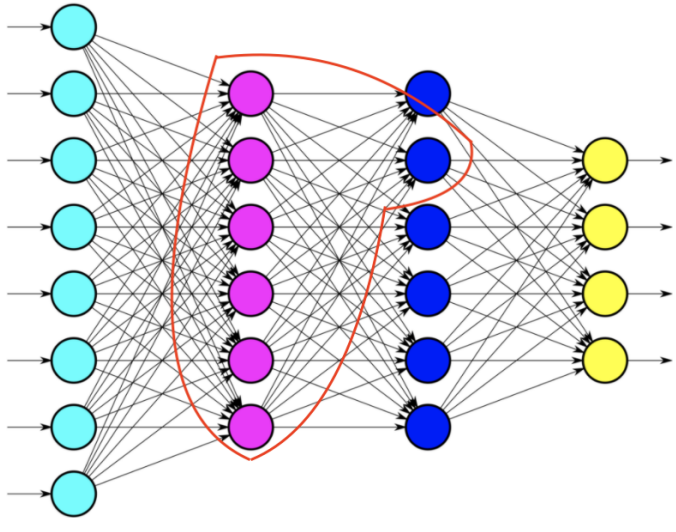
# 완전연결 층의 출력 계산

- 여러 개의 완전연결 층으로 구성된 다층 퍼셉트론 모델의 일반적인 형태

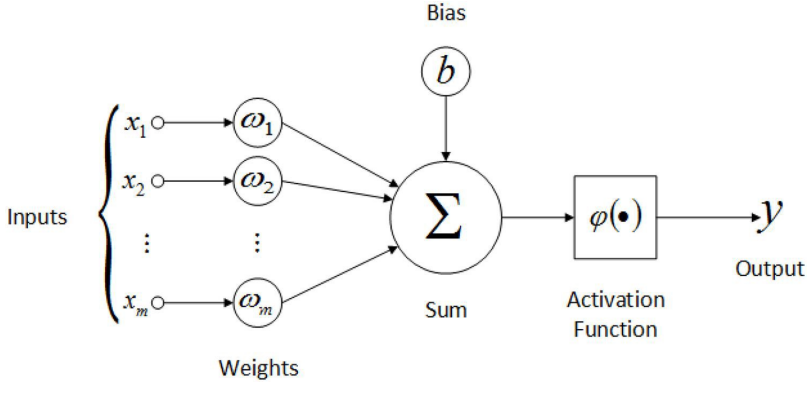


<그림 출처: netclipart (https://www.netclipart.com/down/ihwJhJh_angle-symmetry-area-neural-networks-transparent-backgrounds/)>

- 이 중에서 아래 빨간색으로 이루어진 부분에 대한 계산은 어떻게?

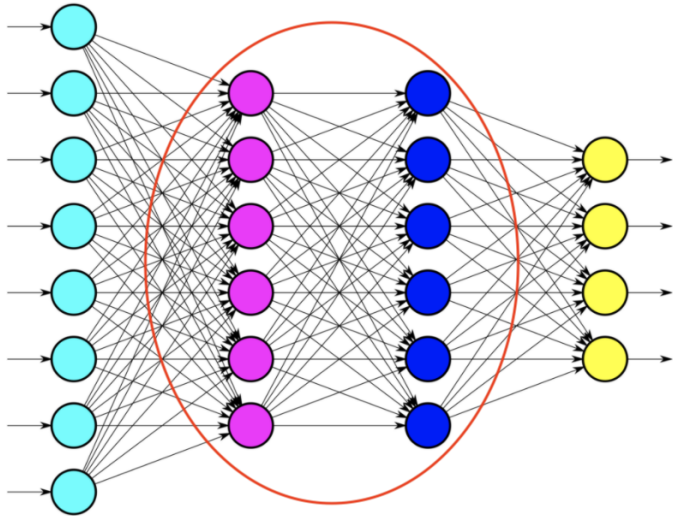


- 다음과 같이 이루어짐.




<그림 출처: medium (https://medium.com/@jayeshbahire/the-artificial-neural-networks-handbook-part-4-d2087d1f583e)>

- 하나의 층에서 이루어지는 입력과 출력을 행렬 수식으로 표현 가능



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

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

- $\mathbf{X}$: 입력 샘플들의 특성행렬.
    - 각 행은 샘플을 가리킴
    - 각 열은 특성을 가리킴

- $\mathbf{W}$: 편향 뉴런을 제외한 모든 뉴런에 대한 가중치.
    - 각 행은 하나의 입력과 연관됨
    - 각 열은 하나의 출력과 연관됨.

- $\mathbf{b}$: 편향벡터
    - 편향 뉴런과 연결된 각각의 출력에 대한 편향값으로 구성된 벡터

- $\phi$: 활성화 함수(activation function)
    - 퍼셉트론 모델처럼 각 인공뉴런이 TLU인 경우, 계단함수가 사용됨.

## 심층신경망(DNN)

- 여러 개의 은닉층을 쌓아올린 인공신경망

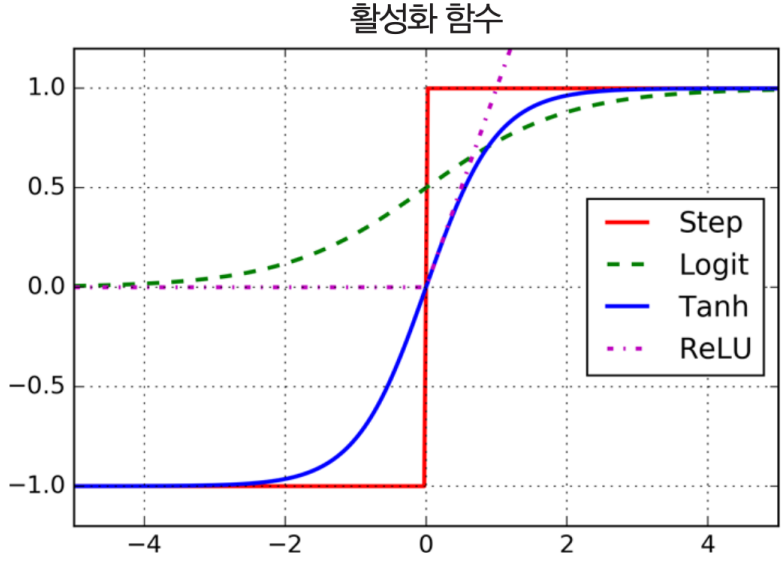
## 역전파 훈련 알고리즘

- 다층 퍼셉트론은 층이 많을 수록 훈련시키는 과정이 점점 더 어려워짐.
- 1986년에 소개된 역전파(backpropagation) 훈련 알고리즘이 발표된 이후로 실용성 갖춤.
- 1단계(정방향): 각 훈련 샘플에 대해 먼저 예측을 만든 후 오차 측정
- 2단계(역방향): 역방향으로 각 층을 거치면서 각 연결이 오차에 기여한 정도 측정
- 3단계: 오차가 감소하도록 모든 가중치 조정

# MLP 특징

- 랜덤하게 설정함. 그렇지 않으면 층의 모든 뉴런이 동일하게 움직임.

- 활성화 함수: 보통 계단함수 대신에 다른 함수 사용.
    - 로지스틱(시그모이드)
    - 하이퍼볼릭 탄젠트 함수(쌍곡 탄젠트 함수)
    - ReLU 함수



## 활성화 함수 대체 필요성

- 선형성을 벗어나기 위해.
    - 선형 변환을 여러 개 연결 해도 선형 변환에 머무름.
    - 따라서 복잡한 문제 해결 불가능.

- 비선형 활성화 함수를 충분히 많은 층에서 사용하면 매우 강력한 모델 학습 가능

# 회귀를 위한 MLP

**출력 뉴런 수**

- 예측해야 하는 값의 수에 따라 출력 뉴런 설정

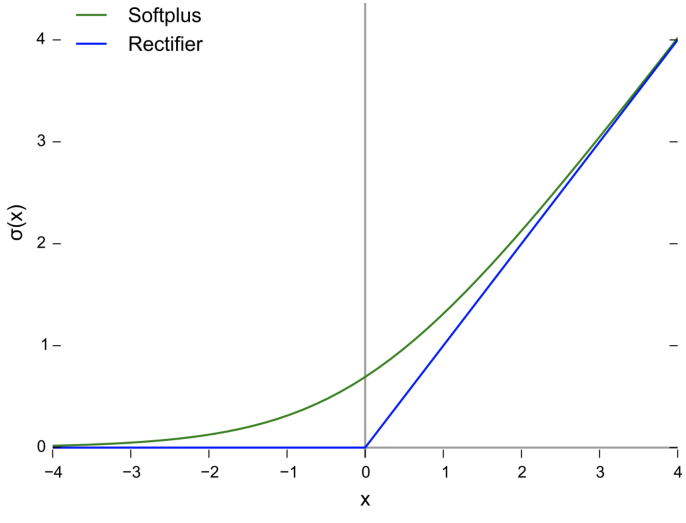
- 예제 1: 주택 가격 예측
    - 출력 뉴런 1개

- 예제 2: 다변량 회귀(동시에 여러값 예측하기)
    - 출력 차원마다 출력 뉴런 1개
    - 예를 들어, 물체의 중심 위치를 알아내려면 좌표 2개 각각에 해당하는 출력 뉴런 2개 필요.

**활성화 함수 지정**

- 출력값에 특별한 제한이 없다면 활성화 함수 사용하지 않음.

- 출력이 양수인 경우
    - ReLU 또는 softplus 사용 가능




<그림 출처: 위키피디아 (https://en.wikipedia.org/wiki/Rectifier_(neural_networks))>

- 출력이 특정 범위 안에 포함되어야 할 경우
    - 로지스틱 함수 또는 하이퍼볼릭 탄젠트 함수와 적절한 스케일 조정 활용

**손실함수**

- 일반적으로 평균제곱오차(MSE) 활용
- 이상치가 많을 경우: 평균절댓값오차(MAE) 사용 가능
- 후버(Huber) 손실 사용 가능
    - MSE와 MAE의 조합

## 회귀 MLP의 전형적인 구조

| 하이퍼파라미터 | 일반적으로 사용되는 값 |
|---|---|
| 입력뉴런 수 | 샘플의 특성마다 하나 |
| 은닉층 수 | 문제에 따라 다름. 보통 1-5개 |
| 은닉층의 뉴런 수 | 문제에 따라 다름. 보통 10-100개 |
| 출력뉴런 수 | 예측 차원마다 하나 |
| 은닉층의 활성화 함수 | ReLU 또는 SELU(11장 참조) |
| 출력층의 활성화 함수 | 보통 없음. 상황에 따라 ReLU/softplus 또는 logistic/tanh 사용 |
| 손실함수 | MSE 또는 MAE/Huber(이상치 많은 경우) |

# 분류를 위한 다층 퍼셉트론

**이진분류**

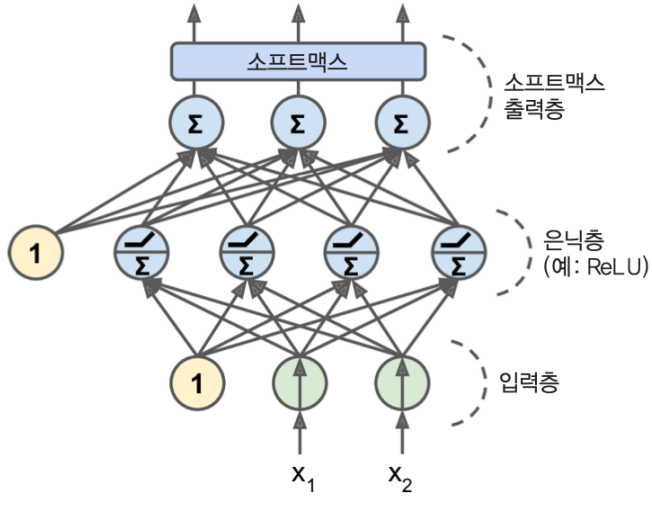
- 하나의 출력 뉴런 사용
- 활성화 함수: 로지스틱 함수

**다중레이블 이진분류**

- 다층 퍼셉트론 활용

- 예제 1: 이메일의 스팸 여부와 함께 긴급메일 여부 확인 가능
    - 활성화 함수: 두 뉴런 모두 로지스틱 함수 사용

- 예제 2: 다중 클래스 분류
    - 3개 이상의 클래스 중 하나의 클래스에만 속해야 하는 경우
    - 예를 들어, MNIST 숫자 이미지. 0부터 9까지.
    - 클래스마다 하나의 뉴런 사용
    - 출력층 활성화 함수: 소프트맥스 함수



**손실함수**

- 크로스 엔트로피(로그 손실). 4장 참조.

## 분류 MLP의 전형적인 구조

| 하이퍼파라미터 | 이진 분류 | 다중레이블 분류 | 다중클래스 분류 |
|---|---|---|---|
| 입력층과 은닉층 | 회귀와 동일 | 회귀와 동일 | 회귀와 동일 |
| 출력뉴런 수 | 1개 | 레이블 당 1개 | 클래스 당 1개 |
| 출력층의 활성화 함수 | 로지스틱 함수 | 로지스틱 함수 | 소프트맥스 함수 |
| 손실함수 | 크로스 엔트로피 | 크로스 엔트로피 | 크로스 엔트로피 |

## 텐서플로 플레이그라운드 활용

- 텐서플로 플레이그라운드 (https://playground.tensorflow.org) 를 이용한 이진 분류기 훈련해 볼 것.

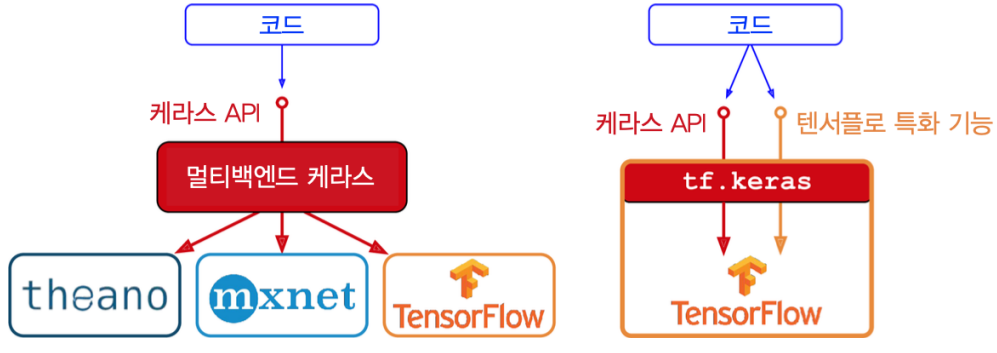
# 케라스(keras)를 이용한 MLP 구현

- 케라스 (https://keras.io)는 모든 종류의 신경망을 손쉽게 만들어 주는 최상위 API 제공.

- 멀티 백엔드 케라스:
    - 구글의 텐서플로(Tensorflow),
    - MS의 Cognitive Toolkit(CNTK),
    - Theano(시애노)
    - 아파치 MXNet
    - 애플 Core ML
    - 자바스크립트, 타입스크립트: 웹브라우저에서 케라스 실행 가능
    - PlaidML: 모든 종류의 GPU에서 실행 가능

## tensorflow.keras

- 텐서플로만 지원하는 keras 백엔드
- 책에서 사용



### 파이토치(PyTorch)

- 파이토치 (https://pytorch.org)는 keras와 비슷한 API 제공하며, 쉽게 배울 수 있음.
- keras만큼 인기 좋음.

# 케라스 Sequential 클래스 활용: 분류

## 패션 MNIST 활용

- 10개의 클래스로 이루어짐.
- 데이터 셋: 28x28 픽셀 크기의 흑백 패션 이미지 샘플 70,000개

## 모델 선언

- `Sequential` 클래스 내에 층을 쌓아 순차적 학습 지원
- 은닉층: 2개

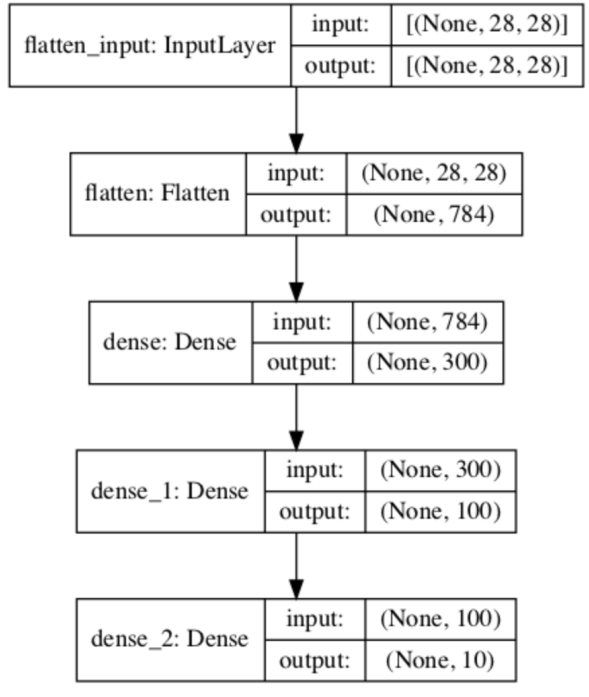
```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

- 아래 방식도 가능

```
model = keras.models.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")
    ])
```

- 아래 명령어를 사용하면 모델 선언된 모델 확인 가능

```
keras.utils.plot_model(model, "my_fashion_mnist_model.png", show_shapes=True)
```


flatten_input: InputLayer
input: [(None, 28, 28)]
output: [(None, 28, 28)]
flatten: Flatten
input: (None, 28, 28)
output: (None, 784)
dense: Dense
input: (None, 784)
output: (None, 300)
dense_1: Dense
input: (None, 300)
output: (None, 100)
dense_2: Dense
input: (None, 100)
output: (None, 10)

## 모델 컴파일하기

- 모델을 생성하려면 선언된 모델을 컴파일 해야함.
- 손실함수, 옵티마이저, 평가기준 등을 지정해야 함.

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

## 모델 훈련과 평가

- 훈련: fit 메서드 호출
- 평가 동시진행 가능

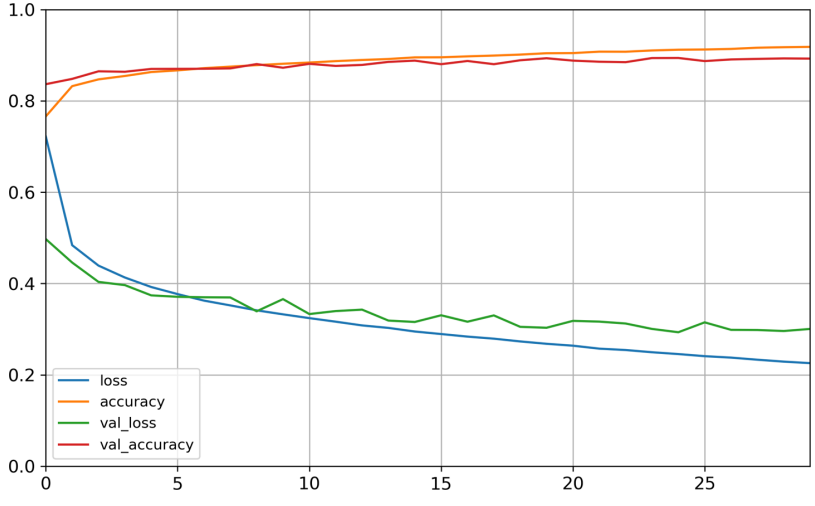
```
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid))
```

## 모델 학습 곡선

- 훈련된 모델이 반환하는 History 객체의 history 속성에 학습된 에포크의 손실과 정확도 기록됨.

```
import pandas as pd

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
save_fig("keras_learning_curves_plot")
plt.show()
```


1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
25
loss
accuracy
val_loss
val_accuracy

## 모델 성능 테스트

- 훈련된 모델의 `evaluate` 메서드 활용
- 손실과 정확도 계산해줌.
- 아래 코드 실행결과: 손실은 0.3339, 정확도는 0.8851

```
model.evaluate(X_test, y_test)
```

**모델 활용**

- 훈련된 모델을 이용한 예측은 `predict` 또는 `predict_class` 메서드 활용.
- `predict` 메서드: 각 클래스에 속할 확률 계산
- `predict_class`: 가장 높은 확률의 클래스 지정

# 케라스 Sequential 클래스 활용: 회귀

**캘리포니아 주택가격 예측**

- `Sequential` 클래스를 이용한 회귀용 MLP 구축은 분류용과 기본적으로 동일.

- 차이점:
    - 출력층에 활성화함수 사용하지 않는 하나의 뉴런만 사용.
        - 일반적으로는 예측값의 수 만큼 출력뉴런 사용
    - 손실함수: 평균제곱오차(MSE)

**캘리포니아 데이터셋과 관련된 주의할 점**

- 잡음이 많음.
- 따라서 과대적합을 줄이기 위해 뉴런 수가 적은 하나의 은닉층만 사용.
- 이유: 은닉층과 뉴런이 많이 사용될 수록 가중치 파라미터의 수가 증가하여 과대적합 위험도 커짐.

## 모델 생성, 훈련 및 평가

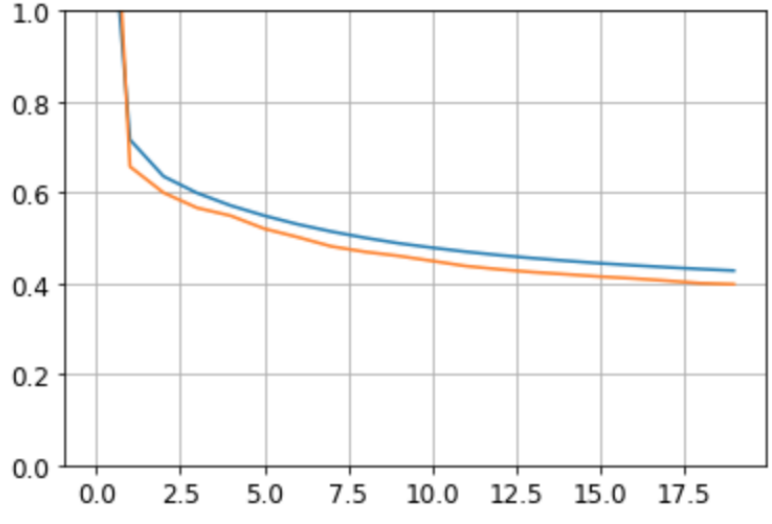
```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
    ])

model.compile(loss="mean_squared_error", optimizer=keras.optimizers.SGD(lr=1e-3))

history = model.fit(X_train, y_train, epochs=20, validation_data=(X_valid, y_vali
d))
```

## 학습 곡선

```
mse_test = model.evaluate(X_test, y_test)
```

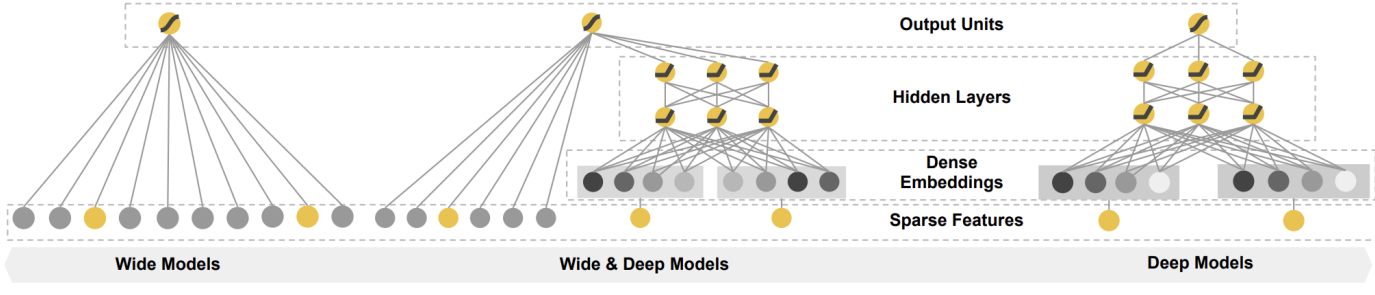


## 케라스 Sequential 클래스 활용의 장단점

- 사용하기 매우 쉬우며 성능 우수함.
- 입출력이 여러 개이거나 더 복잡한 네트워크를 구성하기 어려움.
- `Sequential` 클래스 대신에 **함수형** API, **하위클래스(subclassing)** API 등을 사용하여 보다 복잡하며, 보다 강력한 딥러닝 모델 구축 가능.

# 케라스 함수형 API 활용

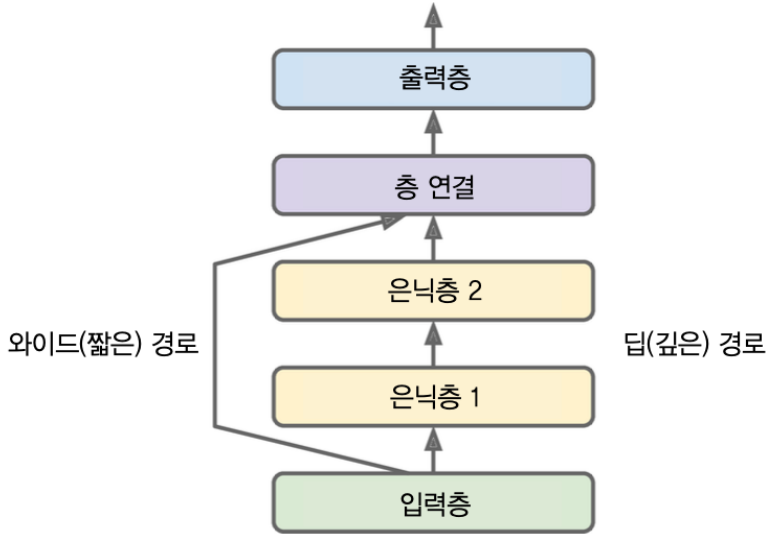
- 모든 레이블을 순차적으로 처리하는 것 대신 다양한 신경망 구축을 위해 함수형 API 활용
- 예제: 와이드&딥(Wide & Deep) 신경망.




<그림 출처: Heng-Tze Cheng et al., "Wide & Deep Learning for Recommender Systems" (https://arxiv.org/pdf/1606.07792.pdf)>

# 와이드&딥 신경망: 활용 1

- 깊게 쌓은 층을 사용한 복잡한 패턴과 짧은 경로를 사용한 간단한 규칙 모두 학습가능한 모델



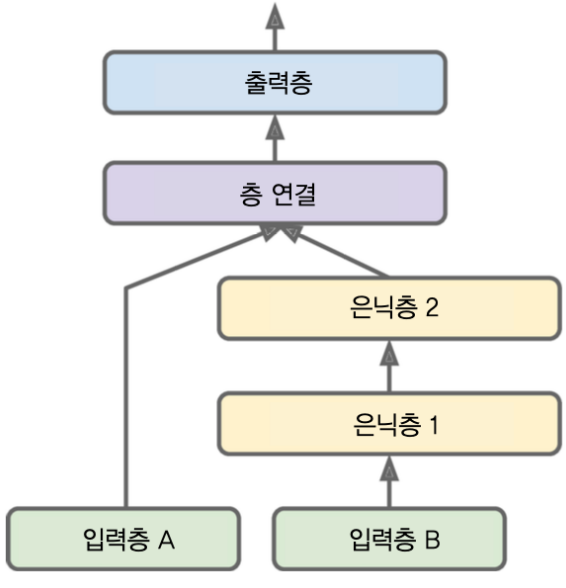
- 예제: 캘리포니아주 주택가격 데이터셋 활용

- `input_`: `Input` 객체
    - `hideen1`을 통해 `hidden2` 로 전달됨
    - 반면에 `concat`를 통해 `hidden2`의 결과와 함께 `output`으로 직접 전달되기도 함.

```
input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.models.Model(inputs=[input_], outputs=[output])
```

# 와이드&딥 신경망: 활용 2

- 다중 입력 사용
    - 특성을 나누어 짧 경로와 깊은 경로에 (중복을 허용하여) 특성을 나누어 보낼 수도 있음.



- `input_A`: 5개의 특성 입력받을 수 있음
- `input_B`: 6개의 특성 입력받을 수 있음
- 입력 뉴런마다 입력값 지정하여 학습해야 함.
    - `evaluate()`, `preict()` 메서드를 호출할 때도 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="output")(concat)
model = keras.models.Model(inputs=[input_A, input_B], outputs=[output])
```

- `input_A` 입력값: 0-4번 인덱스 특성
- `input_B` 입력값: 2-7번 인덱스 특성
- 중복: 2-4번 인덱스 특성

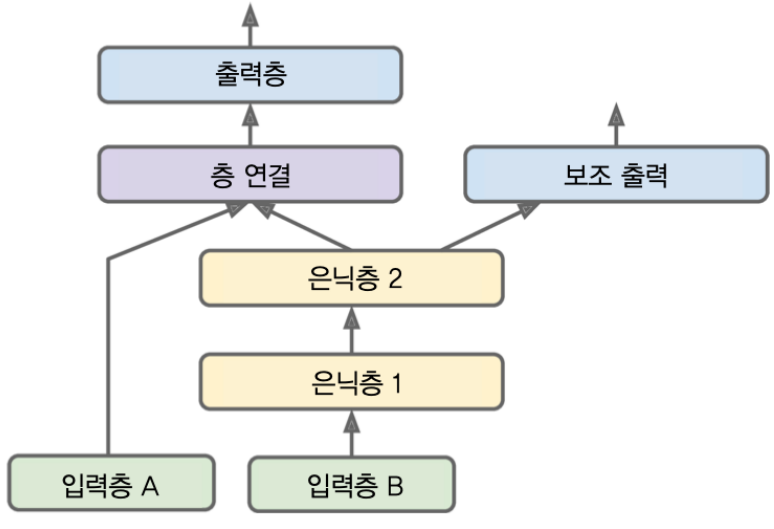
```
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))

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))
```

# 와이드&딥 신경망: 활용 3

- 다중 출력이 필요한 경우 사용



- output: 주 출력
- aux_output: 보조 출력

```
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.models.Model(inputs=[input_A, input_B],
                           outputs=[output, aux_output])
```

- 컴파일할 때 각 출력 뉴런마다 손실함수 지정해야 함
- 출력 별로 **손실가중치** 를 지정하여 출력별 중요도 지정 가능

```
model.compile(loss=["mse", "mse"], loss_weights=[0.9, 0.1],
              optimizer=keras.optimizers.SGD(lr=1e-3))
```

- 모델 훈련 시 출력별로 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]))
```

# 동적 모델 생성: 하위클래스(subclassing) API 활용

- `Sequential` 클래스와 함수형 API 방식을 사용한 모델은 모두 정적임.
    - 한 번 선언되면 변경할 수 없는 모델 생성
    - 모델 저장, 복사, 공유 용이
    - 모델 구조 출력 및 모델 분석 용이

- 반복문, 조건문 등을 활용하여 동적 모델을 생성하고자 할 경우 명령형프로그래밍 방식 요구됨.

- 하위클래스 API을 활용하여 동적 모델 생성 가능

# 하위클래스 API 활용 예제

- `Model` 클래스 상속
    - 초기설정 메서드(`__init__()`)를 이용하여 은닉층과 출력층 설정
    - `call()` 메서드를 이용하여 층을 동적으로 구성 가능
        - for 반복문, if 조건문, 텐서플로 저수준 연산 등을 임의로 활용 가능
    - 참조: 12장

```
class WideAndDeepModel(keras.models.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(30, activation="relu")
```

**단점**

- 모델 구조가 `call()` 메서드 안에 숨겨져 있어서, 케라스가 분석하기 어려움
- 즉, 모델 저장 및 복사 불가능.
- `summary()` 메서드 활용 제한됨: 층의 목록만 확인 가능하며, 층간의 연결정보 알 수 없음.
- 케라스가 타입과 크기를 미리 확인할 수 없어 실수 발생 많아질 수 있음.
- 높은 유연성이 필요하지 않다면 추천하지 않음.

# 모델 저장과 복원

- Sequential 모델과 함수형 API 를 사용해서 훈련된 모델 저장은 매우 간단함.

```
model = keras.models.Sequential([...])
model.compile([...])
model.fit([...])
model.save("my_keras_model.ht")
```

- 저장된 모델을 복원하려면 `load_model()` 함수 활용

```
model = keras.models.load_model("my_keras_model.ht")
```

- 주의사항
    - 서브클래스 API 에서는 사용 불가
    - `save_weights()`, `load_weights()` 메서드를 활용하여 모델 파라미터만 저장/복원 가능
    - 나머지는 수동으로 처리. 예를 들어, `pickle` 모듈 활용.

# 콜백함수 활용

- 훈련의 시작/끝/중간에 호출할 객체를 콜백함수를 이용하여 지정 가능
- 지정된 객체가 호출되면서 각자의 역할 수행

# 예제: ModelCheckpoint

- 훈련 중에 일정 간격으로 모델의 체크포이트 저장
- 체크포인트: 텐서플로우에서 모델 파라미터를 저장하는 양식

```
# 모델 생성 후 훈련하기
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.hs")
history = model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint_cb])
```

- `save_best_only=True` 옵션 설정
  - 훈련 도중에 검증세트를 사용할 경우 최상의 검증 세트 파라미터 저장 가능
- 훈련 종료 후 저장된 최적 모델 복원에 사용

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.hs",
                                                save_best_only=True)
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb])

# 최적 모델 복원
model = keras.models.load_model("my_keras_model.h5")
```

# 예제: EarlyStopping

- 조기종료 구현 용도로 활용
- `patience=10`: 일정 에포크(예를 들어, 10 에포크) 동안 검증세트에 대한 점수가 향상되지 않으면 자동 종료.
    - 에포크 수: 매우 크게 지정해도 됨.
- `restore_best_weights=True`: 최상의 모델 복원 기능 설정. 훈련이 끝난 후 최적 가중치 바로 복원.

```
early_stopping_cb = keras.callbacks.EralyStopping(patience=10,
                                                  restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, early_stopping_cb])
```

## 다양한 콜백함수

- keras.callbacks 패키지가 다양한 콜백함수 제공

# 사용자정의 콜백함수

- keras.callbacks.Callback 클래스 상속
- 상속 과정에서 이미 선언된 콜백함수 메서드 중에서 필요한 메서드를 재정의 하면 됨.
- 예제: 아래 클래스는 에포크가 끝날 때마다 검증손실과 훈련손실의 비율을 계산하고자 할 경우에 필요한 콜백함수 지정

```
class PrintValTrainRatioCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        print("\nval/train: {:.2f}".format(logs["val_loss"] / logs["loss"]))

val_train_ratio_cb = PrintValTrainRatioCallback()
history = model.fit(X_train, y_train, epochs=1,
                    validation_data=(X_valid, y_valid),
                    callbacks=[val_train_ratio_cb])
```

**훈련 단계에서 사용될 수 있는 콜백 함수: (`fit()` 메서드에서 활용됨)**

- on_train_begin()
- on_train_end()
- on_epoch_begin()
- on_epoch_end()
- on_batch_begin()
- on_batch_end()

**평가(검증) 단계에서 사용될 수 있는 콜백 함수: `evaluate()` 메서드에서 활용됨.**

- on_test_begin()
- on_test_end()
- on_test_batch_begin()
- on_test_batch_end()

**예측 단계에서 사용될 수 있는 콜백 함수: `predict()` 메서드에서 활용됨.**

- on_predict_begin()
- on_predict_end()
- on_predict_batch_begin()
- on_predict_batch_end()

# 텐서보드 활용

## 텐서보드

- 대화식 시각화 도구

## 기능

- 훈련하는 동안의 학습곡선,
- 여러 실행 간의 학습곡선 비교,
- 계산그래프 시각화,
- 훈련통계 분석,
- 3D에 사영된 복잡한 다차원 이미지 시각화,
- 자동 군집화, 등등

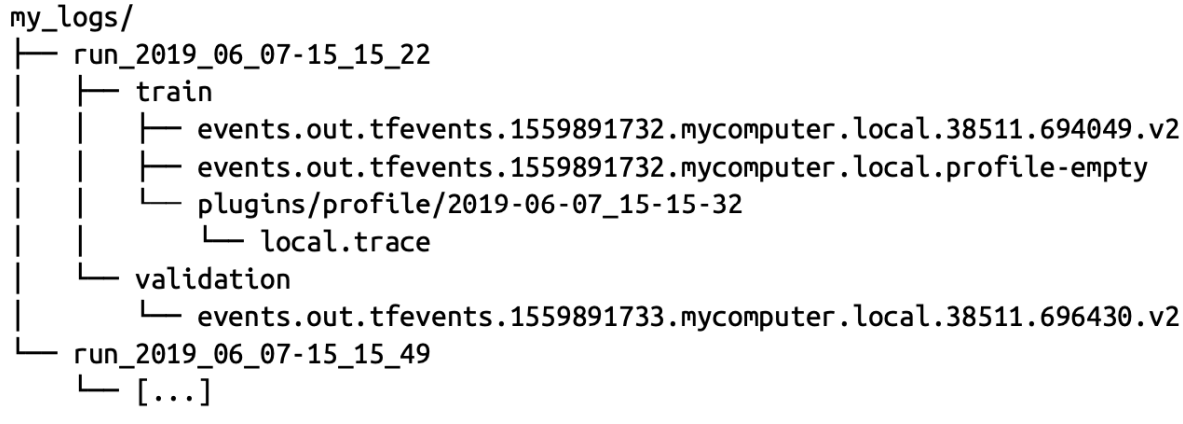
## 사용방식

- 콜백 기능을 이용하여 시각화에 필요한 데이터 저장
- 이벤트 파일: 텐서보드에 활용되는 이진 로그파일
- 로그 디렉토리: 이벤트 파일이 저장되는 디렉토리
- 텐서보드 서버: 로그 디렉토리에 저장된 이벤트 파일을 실시간을로 확인하여 변경사항을 업데이트하는 서버

## 로그 디렉토리 구조

- 루트 로그 디렉토리
    - 모든 로그 디렉토리의 상위 디렉토리
    - 텐서보드 서버가 실시간으로 감시하는 디렉토리

- 하위 로그 디렉토리
    - 훈련할 때마다 새로운 하위 로그 디렉토리를 생성하여 그곳에 이벤트 파일 저장
    - 여러 번 실행한 훈련 결과를 시각화하고 비교 가능해짐.


my_logs/
├── run_2019_06_07-15_15_22
│   ├── train
│   │   ├── events.out.tfevents.1559891732.mycomputer.local.38511.694049.v2
│   │   ├── events.out.tfevents.1559891732.mycomputer.local.profile-empty
│   │   └── plugins/profile/2019-06-07_15-15-32
│   │       └── local.trace
│   └── validation
│       └── events.out.tfevents.1559891733.mycomputer.local.38511.696430.v2
└── run_2019_06_07-15_15_49
    └── [...]

## 예제

- 루트 로그 디렉토리 지정
- `get-run_logdir()` 함수
    - 훈련 실행할 때 사용될 하위 로그 디렉토리 생성
    - 현재 시간 활용하여 매번 새로운 디렉토리 이름 설정

```
# 루트 로그 디렉토리 지정
root_logdir = os.path.join(os.curdir, "my_logs")

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()
run_logdir
```

- 모델 구성과 컴파일

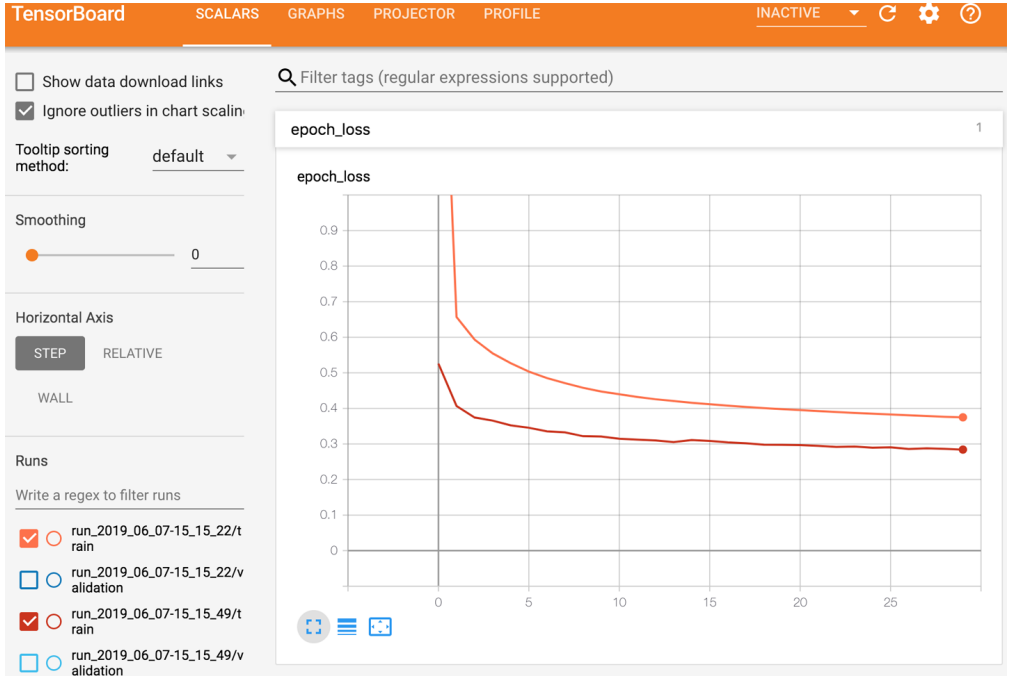
```
# 텐서보드 콜백함수 지정
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)

history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, tensorboard_cb])
```

- 텐서보드 실행 (명령 터미널에서 실행)

```
$ tensorboard --logdir=./my_logs --port=6006
```

- 인터넷 브라우저에서 `http://localhost:6006` 열면 텐서보드 실행 확인 가능



# 신경망 하이퍼파라미터 세부조정

- 신경망은 조정해야할 하이퍼파라미터가 매우 많음.
- 복잡한 네트워크 구조
- 층과 뉴런의 수
- 활성화 함수, 가중치 초기화 등등

# 그리드탐색/랜덤탐색 활용

- 케라스 모델을 사이킷런의 추정기처럼 보여주어야 함.

- 회귀 예제: `KerasRegressor` 클래스를 이용하여 케라스 모델을 사이킷런 모델처럼 작동하게 만들 수 있음.

- `KerasRegressor` 클래스의 객체를 생성할 때 아래 `build_model()` 함수와 같이 케라스 모델을 생성하는 함수를 입력해 주어야 함.

## 케라스 모델 생성함수 예제

```
def build_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(lr=learning_rate)
    model.compile(loss="mse", optimizer=optimizer)
    return model
```

## KerasRegressor 클래스 활용 예제

```
keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)
```

## 랜덤탐색 활용 예제

```
param_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, param_distribs, n_iter=10, cv=3, ve
rbose=2)
rnd_search_cv.fit(X_train, y_train, epochs=100,
                  validation_data=(X_valid, y_valid),
                  callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

# 더 좋은 기법

- 랜덤탐색 기법은 매우 오래 걸림.

- 하이퍼파라미터 최적화에 사용할 수 있는 다양한 라이브러리 개발됨.
    - Hyperopt
    - Hyperas, kopt, Talos
    - Keras Tuner
    - Scikit-Optimize(skopt)
    - Spearmint
    - Hyperband
    - Sklearn-Deap

- 클라우드 서비스도 활용 가능
    - 구글의 AutoML

# 하이퍼파라리터 조정 가이드라인

### 은닉층 개수

- 복잡한 문제 해결의 경우: 심층 신경망이 효율적임.
- 과대적합이 발생할 때까지 은닉층 수를 늘려가며 훈련시킬 수 있음.
- 수십, 수백층이 필요한 경우도 있음. 하지만 밀집연결이 사용되진 않음. (14장 참조)
- 수십, 수백층의 신경망을 처음부터 훈련시키는 대신 훈련된 네트워크를 재사용하는 것이 일반적임.

**은닉층의 뉴런 수**

- 일단 필요한 것보다 더 많은 층과 뉴런을 가진 모델로 시작
- 과대적합되지 않도록 조기종료나 규제 기법 사용 추천

**학습률**

- $10^{-5}$ 에서 시작하여 10까지 조정해보며 실험
- 변경 크기는 두 구간의 크기는 500으로 나눈 정도.
    - 약, $\exp(\log(10^6)/500$ 정도씩 크게해줌.

**옵티마이저**

- 고전적인 미니배치 경사하강법 보다 더 좋은 성능의 옵티마이저 사용 가능
- 11장 참조

**배치크기**

- 경우에 따라 다름.
- 일반적으로 2-32.
- 어떨 때는 8192까지 가능.
- 훈련된 성능이 좋지 않을 경우 작은 배치크기 추천.

**활성화함수**

- 은닉층: ReLU 가 일반적.
- 출력층: 모델에 따라 다름.

**반복횟수(에포크)**

- 조기종료 기법을 사용 추천

## 학습률의 의존성

- 최적의 학습률은 다른 하이퍼파라미터에 의존적임.
- 특히 배치크기에 영양 많이 받음.
- 따라서 다른 하이퍼파라이머터를 조정한 경우, 학습률도 조정해야 함.