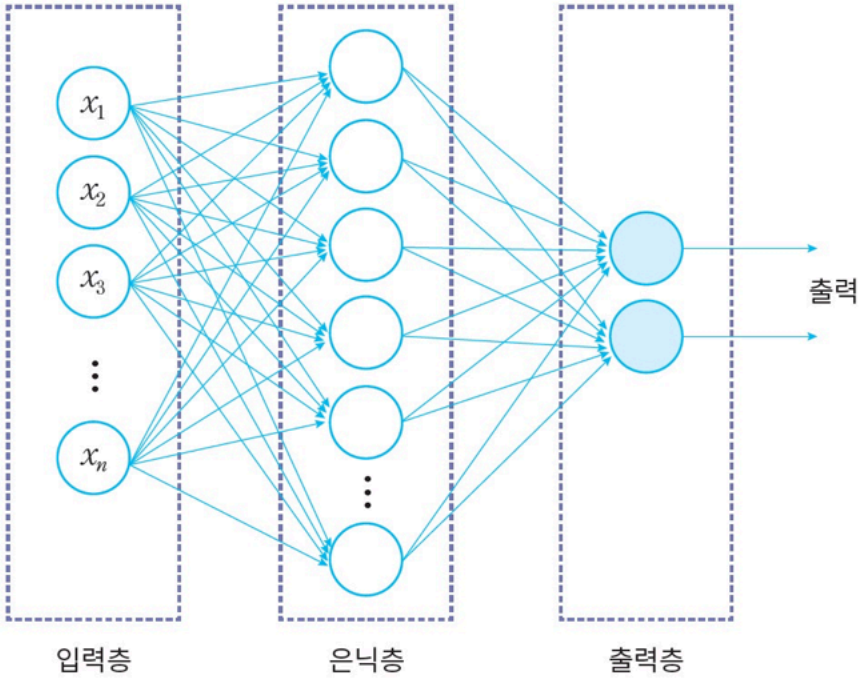
Neural Network

학습 목표

- MLP의 작동 원리
- MLP 학습 알고리즘 이해
- MLP를 구현

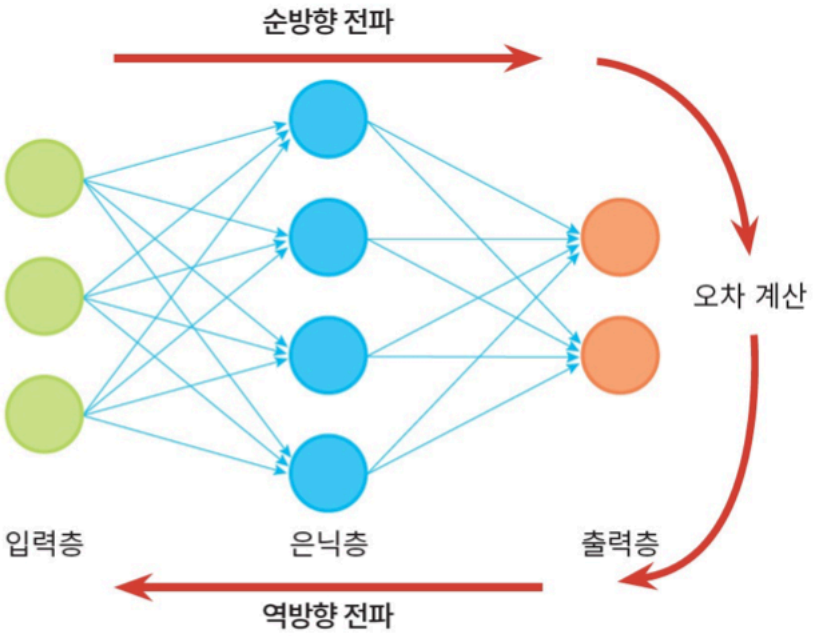
MLP

- 다층 퍼셉트론, multilayer perceptron (MLP)
- 입력층과 출력층 사이에 은닉층(hidden layer)을 가지고 있는 신경망



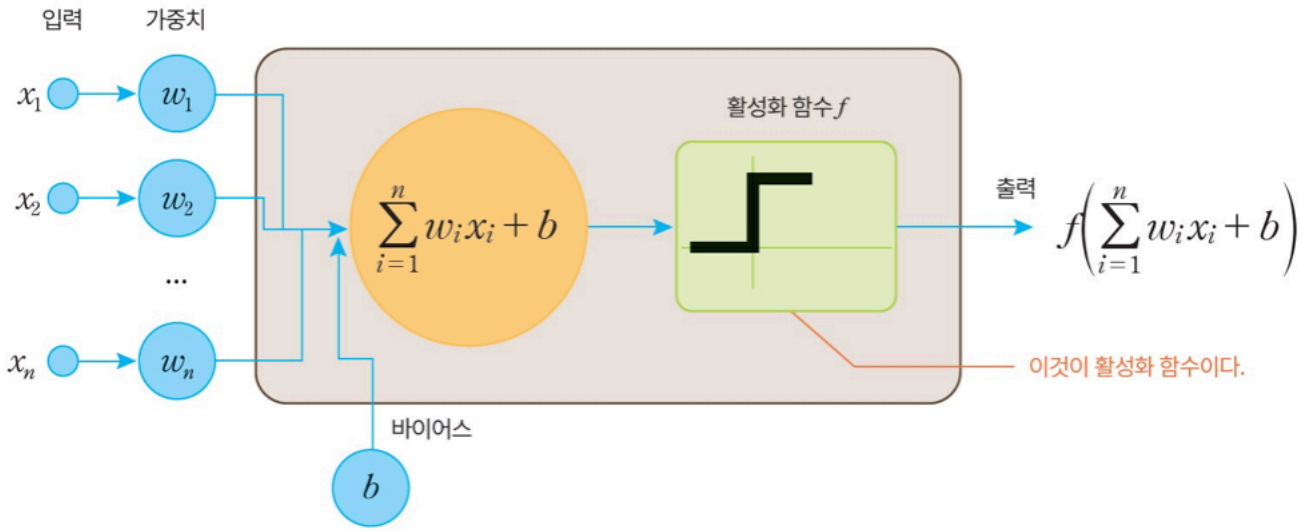
역전파 학습 알고리즘

- 역전파 알고리즘은 입력이 주어지면 순방향으로 계산하여 출력을 계산한 후에 실제 출력과 우리가 원하는 출력 간의 오차를 계산한다.
- 이 오차를 역방향으로 전파하면서 오차를 줄이는 방향으로 가중치를 변경한다.



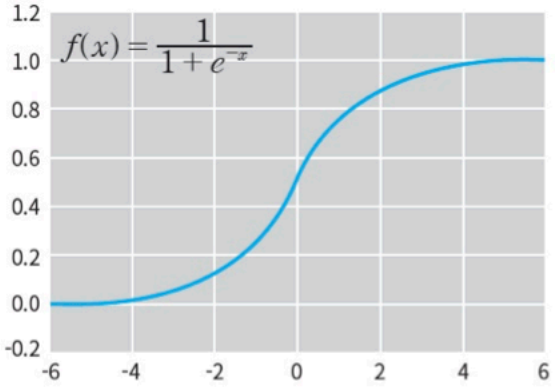
활성화 함수

- 퍼셉트론에서는 계단 함수(step function)를 활성화 함수로 사용
- MLP에서는 다양한 비선형 함수들을 활성화 함수로 사용

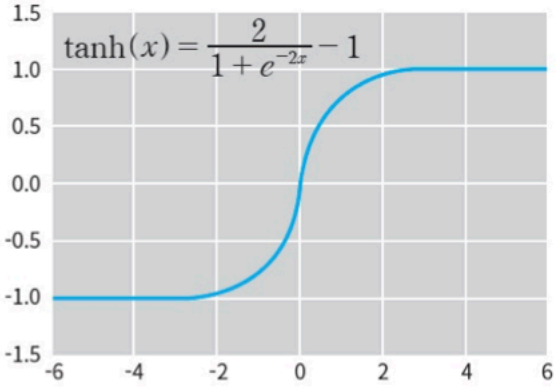


활성화 함수

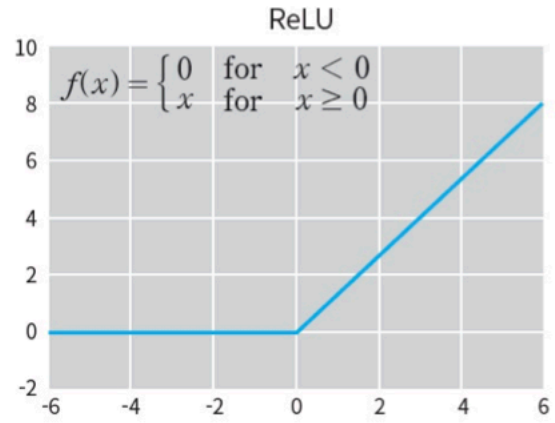
Sigmoid



TanH



ReLU



역전파 학습 알고리즘

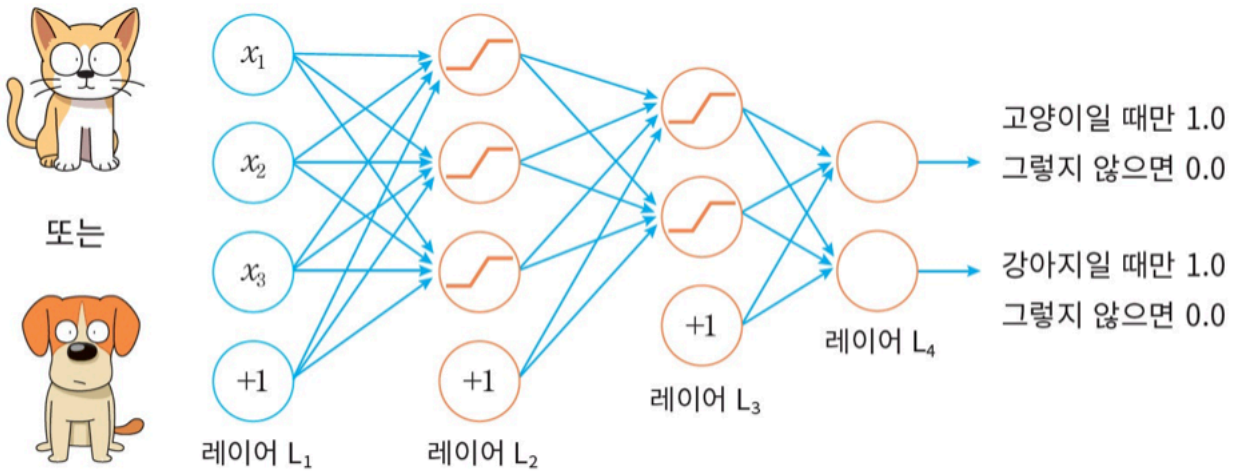
- 순방향 출력 계산

```
current = input
for each layer in network
  for each neuron i in layer
          net = get_sum(weights[i], current)       # 입력의 가중합을 계산
          output[i] = activation_func(net)          # 각 노드들의 출력을 계산한다.
  current = output                # 다음 계층은 현재 계층의 출력을 입력으로 사용한다.
```

손실 함수

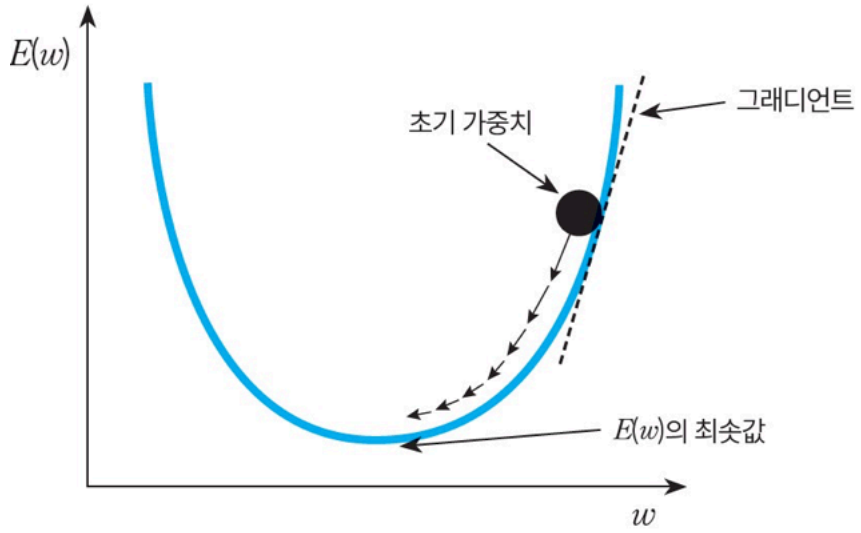
- 전체 오차는 목표 출력값과 실제 기대값과 차를 제곱한 값을 모든 출력 노드에 대하여 합산

$$E(w) = \frac{1}{2}\sum_i (t_i - o_i)^2$$



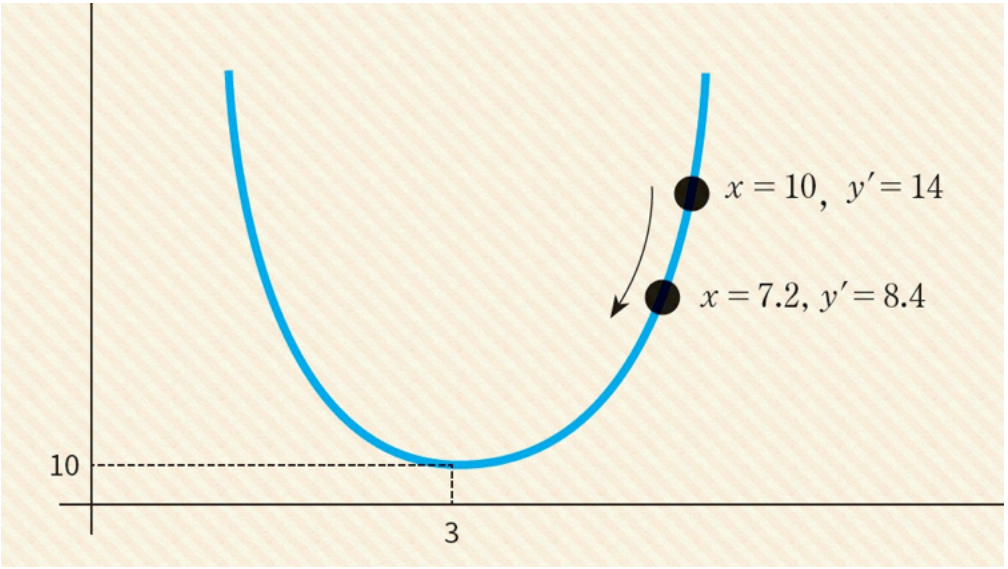
경사 하강법

- 현재 가중치를 함수의 그래디언트값을 계산한 후에 그래디언트의 반대 방향으로 조정



Lab: 경사하강법의 실습

- 손실 함수 $y = (x-3)^2 + 10$
- 그래디언트: y' = 2x-6



경사 하강법 프로그래밍

```
x = 10
learning_rate = 0.01
precision = 0.00001
max_iterations = 100

# 손실 함수를 람다식으로 정의한다.
loss_func = lambda x: (x-3)**2 + 10

# 그래디언트를 람다식으로 정의한다. 손실 함수의 1차 미분값이다.
gradient = lambda x: 2*x-6

# 그래디언트 강하법
for i in range(max_iterations):
    x = x - learning_rate * gradient(x)
    print("손실 함수값(", x, ")=", loss_func(x))

print("최소값 = ", x)
```

실행 결과

```
...
손실 함수값( 4.02701132062644 )= 11.054752252694865
손실 함수값( 4.006471094213911 )= 11.012984063488148
손실 함수값( 3.9863416723296328 )= 10.972869894574016
손실 함수값( 3.9666148388830402 )= 10.934344246748886
손실 함수값( 3.947282542105379 )= 10.897344214577629
손실 함수값( 3.9283368912632715 )= 10.861809383680356
최소값 =  3.9283368912632715
```

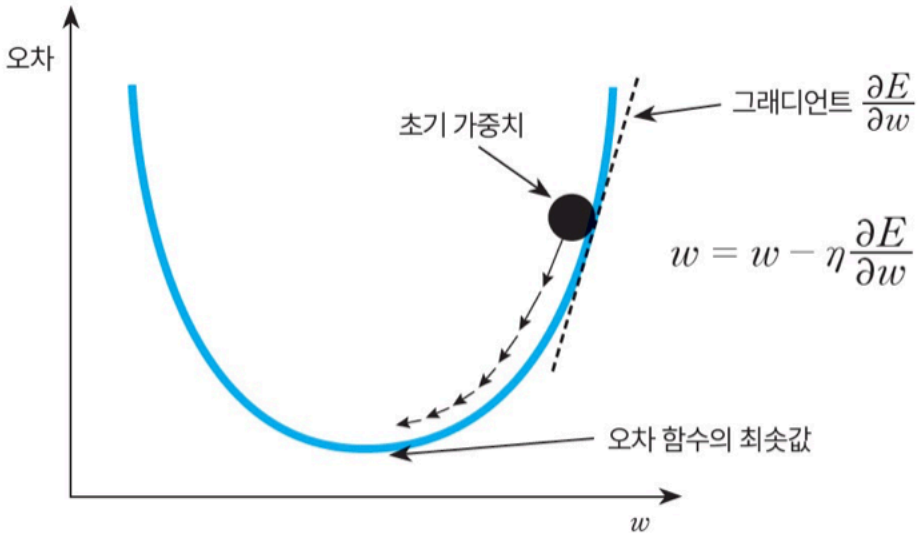
역전파 알고리즘의 유도

- 손실 함수 $E(w) = \frac{1}{2}\sum_i (t_i - o_i)^2$

$$w(t+1) = w(t) - \eta \frac{\partial E}{\partial w}$$

- 가중치의 변경:

$$\frac{\partial E}{\partial w}$$



역전파 알고리즘 의사 코드

```
신경망의 가중치를 작은 난수로 초기화한다.
do 각 훈련 샘플 sample에 대하여 다음을 반복한다.
        actual = calculate_network(sample)  // 순방향 패스
        target = desired_output(sample)
        각 출력 노드에서 오차(target - actual)을 계산한다.
        은익층에서 출력층으로의 가중치 변경값을 계산한다. // 역방향 패스
        입력층에서 은닉층으로의 가중치 변경값을 계산한다. // 역방향 패스
       전체 가중치를 업데이트한다.
until 모든 샘플이 올바르게 분류될 때까지
```

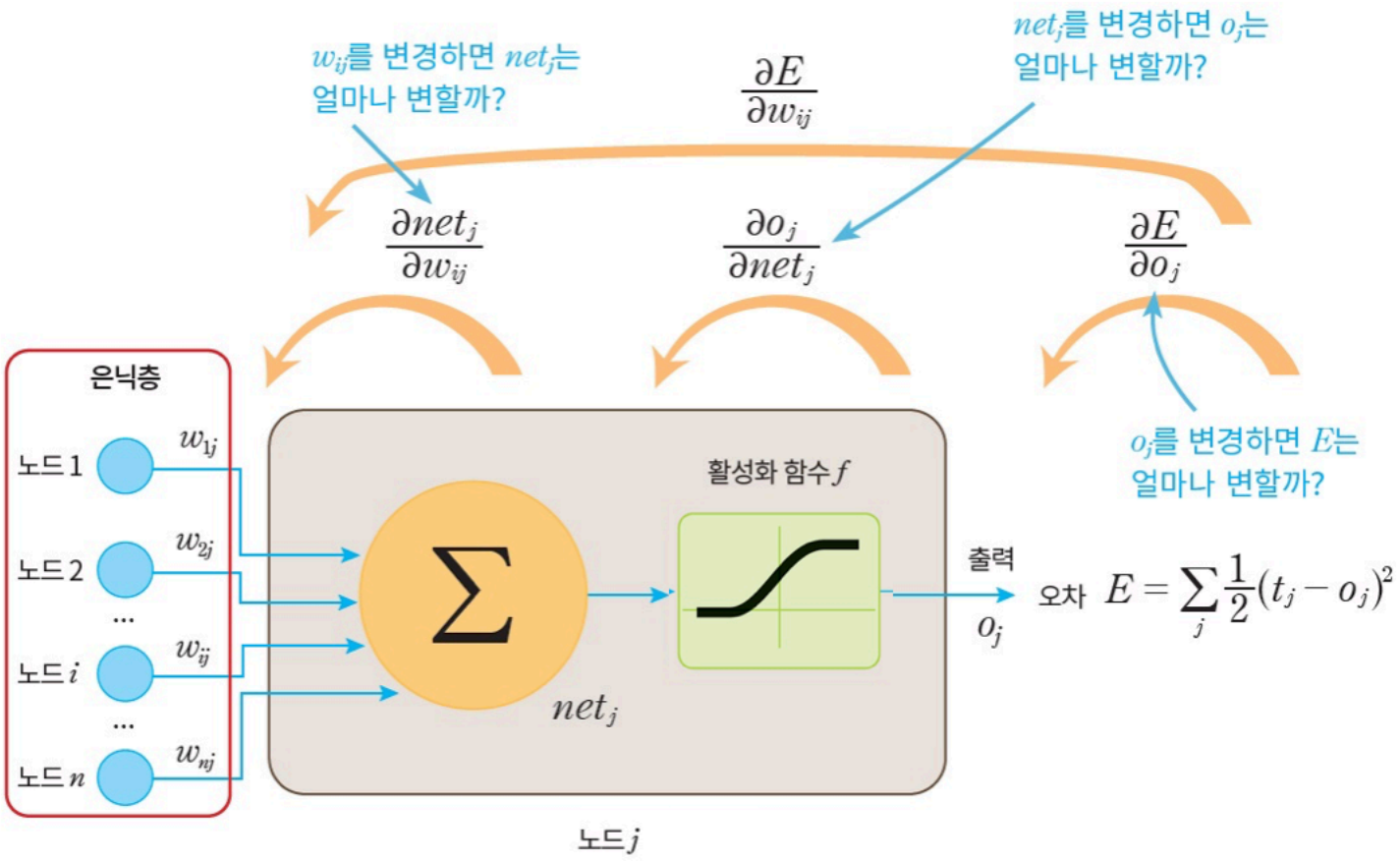
역전파 알고리즘의 설명

- Chain rule 이용

$$\frac{\partial E}{\partial w_{ij}} = \boxed{\frac{\partial E}{\partial o_j}} \boxed{\frac{\partial o_j}{\partial net_j}} \boxed{\frac{\partial net_j}{\partial w_{ij}}}$$

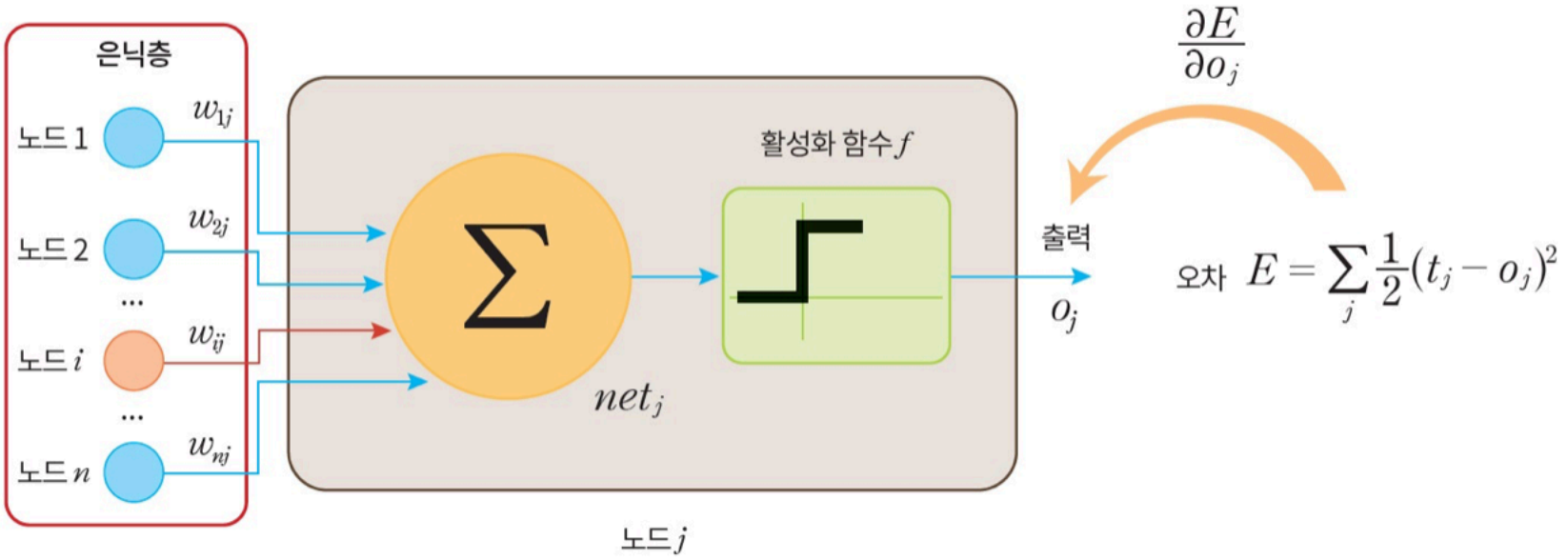
1 2 3

역전파 알고리즘의 설명


w_{ij}를 변경하면 net_j는 얼마나 변할까?
$\frac{\partial E}{\partial w_{ij}}$
net_j를 변경하면 o_j는 얼마나 변할까?
$\frac{\partial net_j}{\partial w_{ij}}$
$\frac{\partial o_j}{\partial net_j}$
$\frac{\partial E}{\partial o_j}$
은닉층
노드 1
노드 2
노드 i
노드 n
w_{1j}
w_{2j}
w_{ij}
w_{nj}
활성화 함수 f
Σ
net_j
출력
o_j
o_j를 변경하면 E는 얼마나 변할까?
오차 $E = \sum_j \frac{1}{2}(t_j - o_j)^2$
노드 j

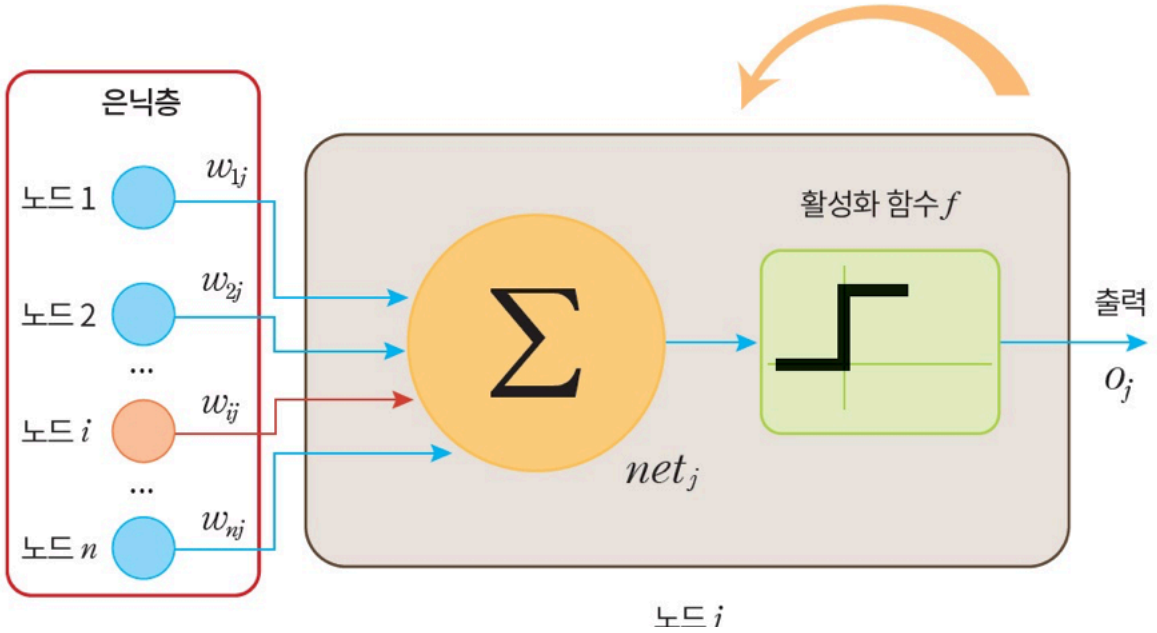
역전파 알고리즘의 설명

$$\frac{\partial E}{\partial o_j} = \frac{\partial}{\partial o_j}\sum_k \frac{1}{2}(t_k - o_k)^2 = o_j - t_j$$



역전파 알고리즘의 설명

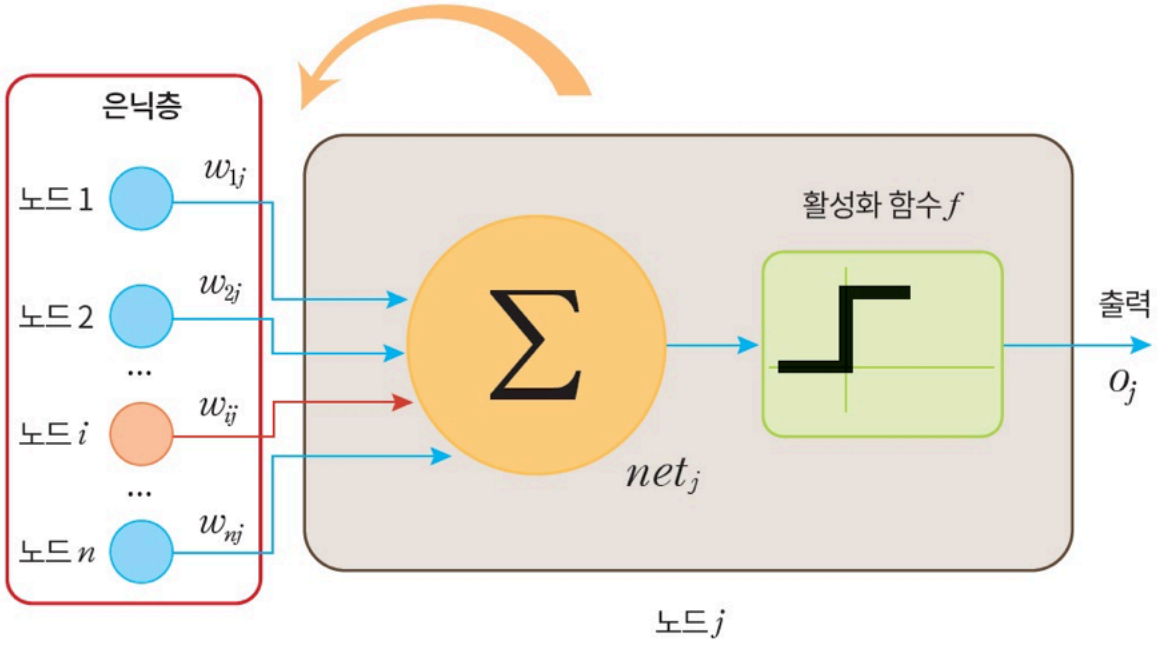
$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \varphi(net_j)}{\partial net_j} = \varphi(net_j)(1-\varphi(net_j))$$



오차 $E = \sum_j \frac{1}{2}(t_j - o_j)^2$

역전파 알고리즘의 설명

$$\frac{\partial \mathrm{net}_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{k=0}^{n} w_{kj} o_k\right) = \frac{\partial}{\partial w_{ij}} w_{ij} o_i = o_i$$



오차 $E = \sum_j \frac{1}{2}(t_j - o_j)^2$

가중치 update

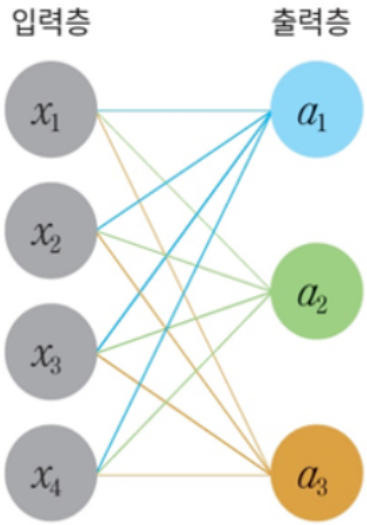
$$\delta_j = \begin{cases} \varphi'(\mathrm{net_j})(\mathrm{o_j} - \mathrm{t_j}) & \text{j가 출력 계층이면} \\ (\sum_{\mathrm{k}} \mathrm{w_{jk}}\delta_{\mathrm{k}})\varphi'(\mathrm{net_j}) & \text{j가 은닉 계층이면} \end{cases}$$

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = -\eta o_i \delta_j$$

넘파이를 이용하여 MLP 구현

- 넘파이의 기능을 이용하면 모든 것을 행렬과 벡터로 표시
- 행렬을 이용하면 동시에 여러 개의 예제를 동시에 학습 가능
- 역전파할 때는 가중치 행렬를 전치시켜서 사용
- 바이어스는 입력을 1.0으로 고정하고, 이 입력에 붙은 가중치로 고려

넘파이를 이용하여 MLP 구현

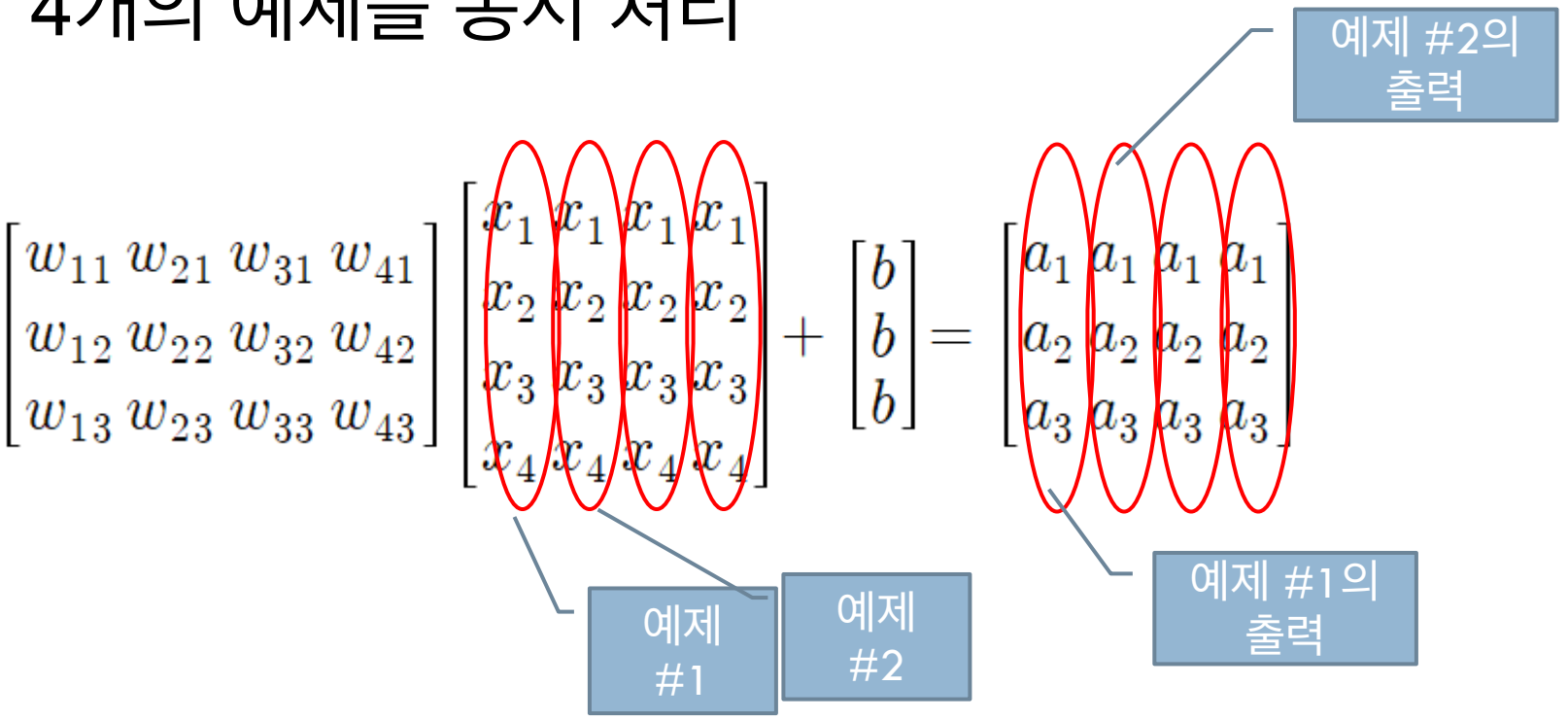


$$\begin{bmatrix} w_{11} & w_{21} & w_{31} & w_{41} \\ w_{12} & w_{22} & w_{32} & w_{42} \\ w_{13} & w_{23} & w_{33} & w_{43} \end{bmatrix} \begin{bmatrix} x_1 & x_1 & x_1 & x_1 \\ x_2 & x_2 & x_2 & x_2 \\ x_3 & x_3 & x_3 & x_3 \\ x_4 & x_4 & x_4 & x_4 \end{bmatrix} + \begin{bmatrix} b \\ b \\ b \end{bmatrix} = \begin{bmatrix} a_1 & a_1 & a_1 & a_1 \\ a_2 & a_2 & a_2 & a_2 \\ a_3 & a_3 & a_3 & a_3 \end{bmatrix}$$

- 첫 번째 예제에 대하여 출력 노드 a_1 값을 계산

$$w_{11}{}^*x_1 + w_{21}{}^*x_2 + w_{31}{}^*x_3 + w_{41}{}^*x_4 + b = a_1$$

- 4개의 예제를 동시 처리



소스

```
import numpy as np

# 시그모이드 함수
def actf(x):
  return 1/(1+np.exp(-x))

# 시그모이드 함수의 미분값
def actf_deriv(x):
      return x*(1-x)

# XOR 연산을 위한 4행*2열의 입력 행렬
# 마지막 열은 바이어스를 나타낸다.
X = np.array([[0,0,1], [0,1,1], [1,0,1], [1,1,1]])

# XOR 연산을 위한 4행*1열의 목표 행렬
y = np.array([[0], [1], [1], [0]])
```

소스

```
np.random.seed(5)

inputs = 3              # 입력층의 노드 개수
hiddens = 6             # 은닉층의 노드 개수
outputs = 1             # 출력층의 노드 개수

# 가중치를 -1.0에서 1.0 사이의 난수로 초기화한다.
weight0 = 2*np.random.random((inputs, hiddens))-1
weight1 = 2*np.random.random((hiddens, outputs))-1

# 반복한다.
for i in range(10000):

    # 순방향 계산
    layer0 = X                                          # 입력을 layer0에 대입한다.
    net1 = np.dot(layer0, weight0)        # 행렬의 곱을 계산한다.
    layer1 = actf(net1)                               # 활성화 함수를 적용한다.
    layer1[:,-1] = 1.0              #  마지막 열은 바이어스를 나타낸다. 1.0으로 만든다.
    net2 = np.dot(layer1, weight1)        # 행렬의 곱을 계산한다.
    layer2 = actf(net2)                               # 활성화 함수를 적용한다.
```

소스

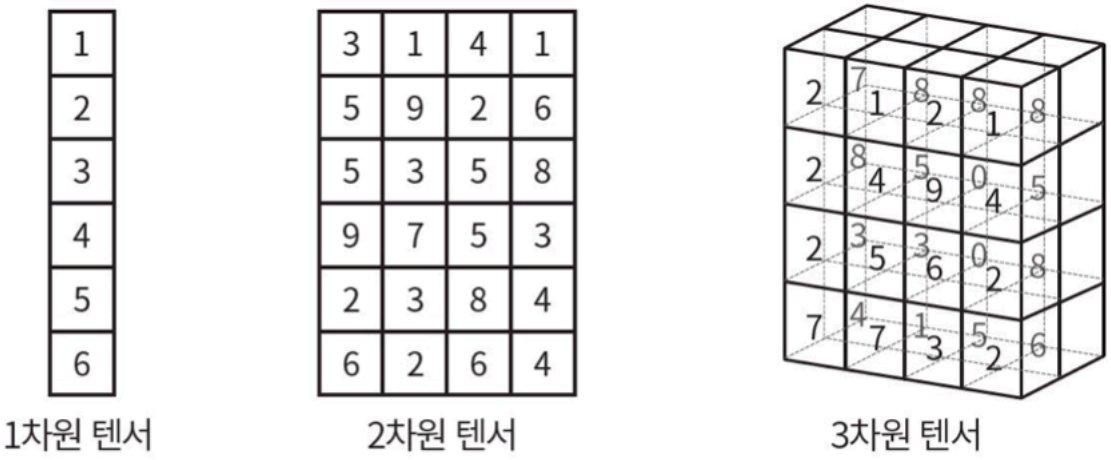
```
 # 출력층에서의 오차를 계산한다.
    layer2_error = layer2-y

    # 출력층에서의 델타값을 계산한다.
    layer2_delta = layer2_error*actf_deriv(layer2)

    # 은닉층에서의 오차를 계산한다.
    # 여기서 T는 행렬의 전치를 의미한다.
    # 역방향으로 오차를 전파할 때는 반대방향이므로 행렬이 전치되어야 한다.
    layer1_error = np.dot(layer2_delta, weight1.T)

    # 은닉층에서의 델타를 계산한다.
    layer1_delta = layer1_error*actf_deriv(layer1)

    # 은닉층->출력층을 연결하는 가중치를 수정한다.
    weight1 += -0.2*np.dot(layer1.T, layer2_delta)

    # 입력층->은닉층을 연결하는 가중치를 수정한다.
    weight0 += -0.2*np.dot(layer0.T, layer1_delta)
print(layer2)                                   # 현재 출력층의 값을 출력한다.
```

실행결과

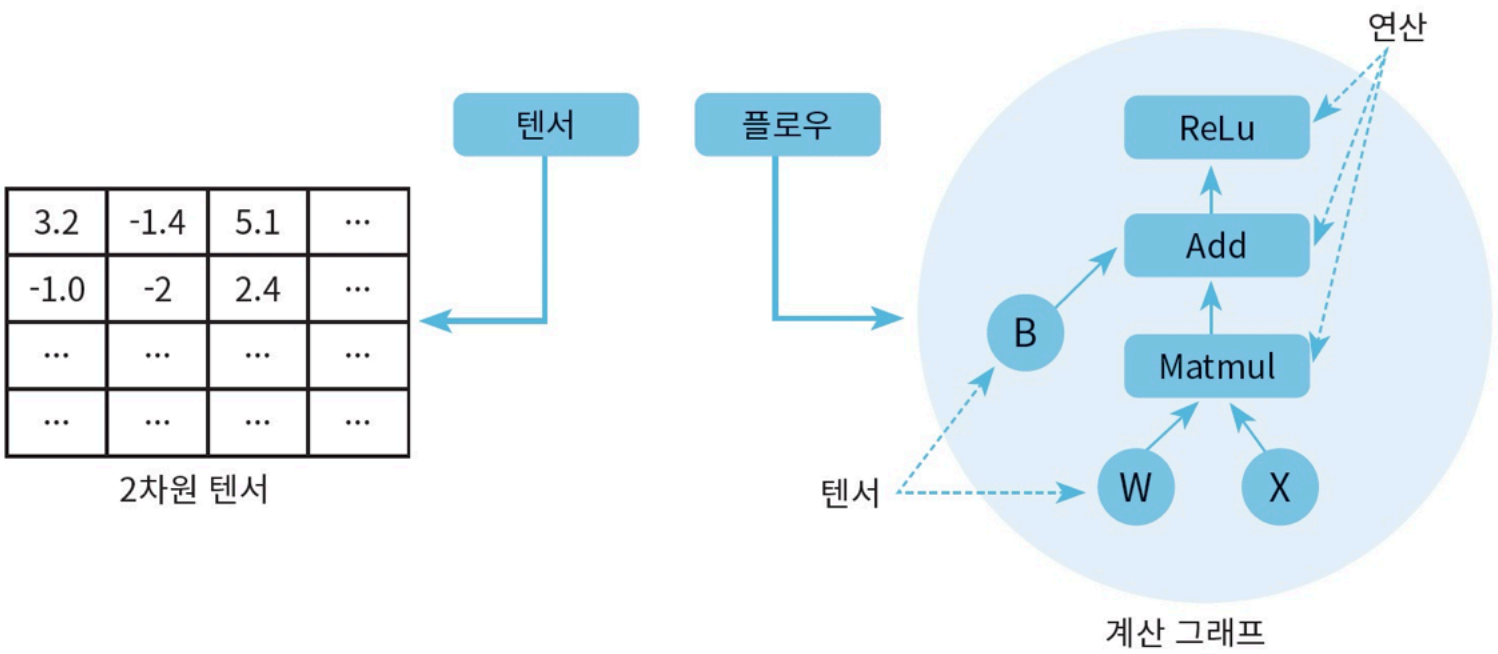
```
[[0.02391914]
 [0.9757925 ]
 [0.97343127]
 [0.03041428]]
```

구글의 텐서플로우

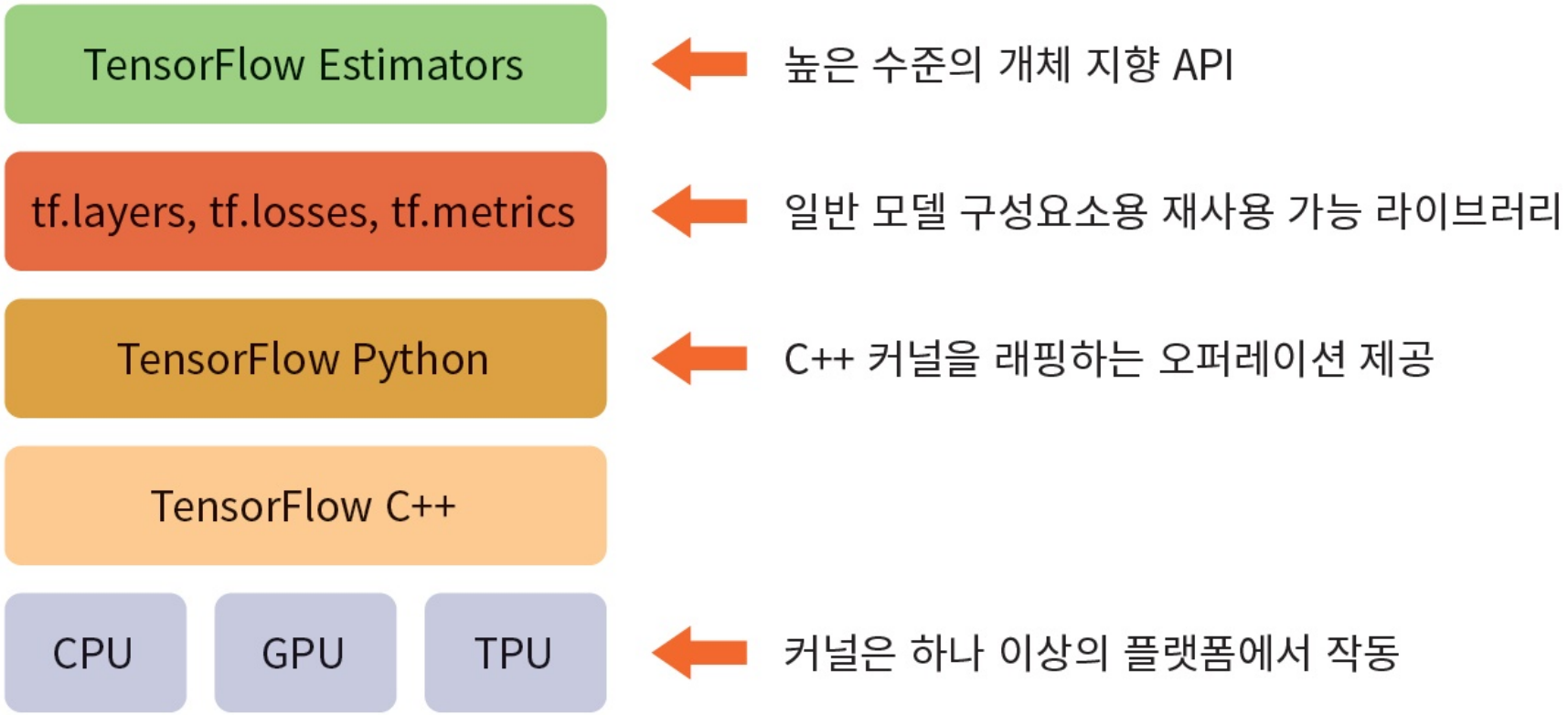
- 텐서플로우(TensorFlow)는 딥러닝 프레임워크의 일종
- 텐서플로우는 내부적으로 C/C++로 구현
- Python을 포함하는 여러 가지 언어에서 접근할 수 있도록 인터페이스를 제공



구글의 텐서플로우



구글의 텐서플로우



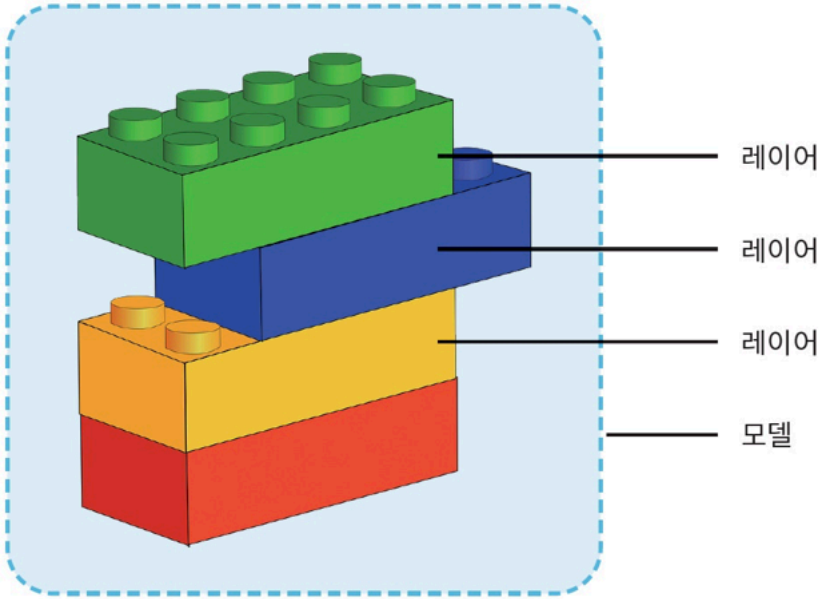
(그림 출처: 구글)

Keras

- Keras는 Python으로 작성되었으며 TensorFlow , CNTK 또는 Theano에서 실행할 수 있는 고수준 딥러닝 API
  - 쉽고 빠른 프로토타이핑이 가능
  - 순방향 신경망, 컨볼루션 신경망과 반복적인 신경망은 물론 물론 여러 가지의 조합도 지원
  - CPU 및 GPU에서 실행

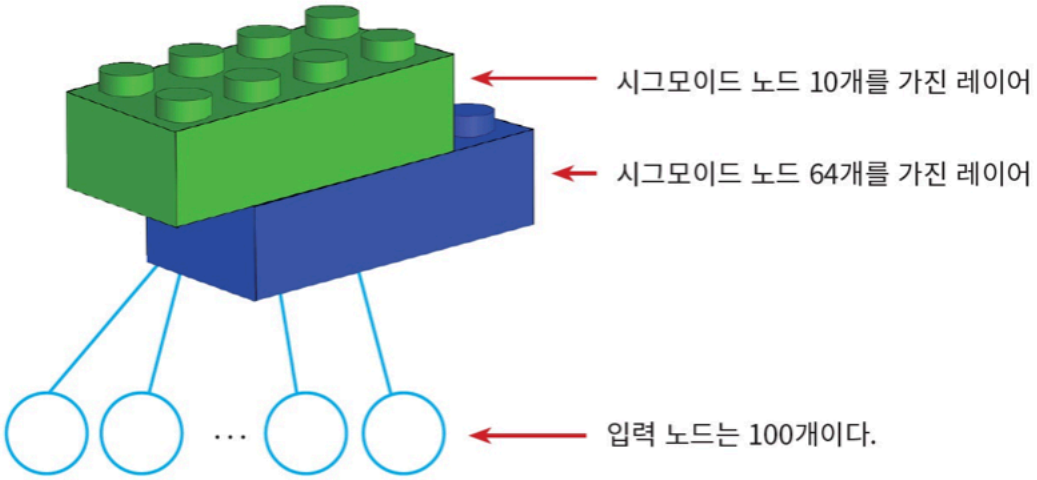
Keras

- Keras는 신경망을 레고 조립하듯이 모델 생성



모델 작성

```
from tf.keras.models import Sequential

model = Sequential()

from tf.keras.layers import Dense

model.add(Dense(units=64, activation='sigmoid', input_dim=100))   # ①
model.add(Dense(units=10, activation='sigmoid'))                  # ②
```



학습 과정 정의

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

model.fit(X, y, epochs=5, batch_size=32)
```

학습

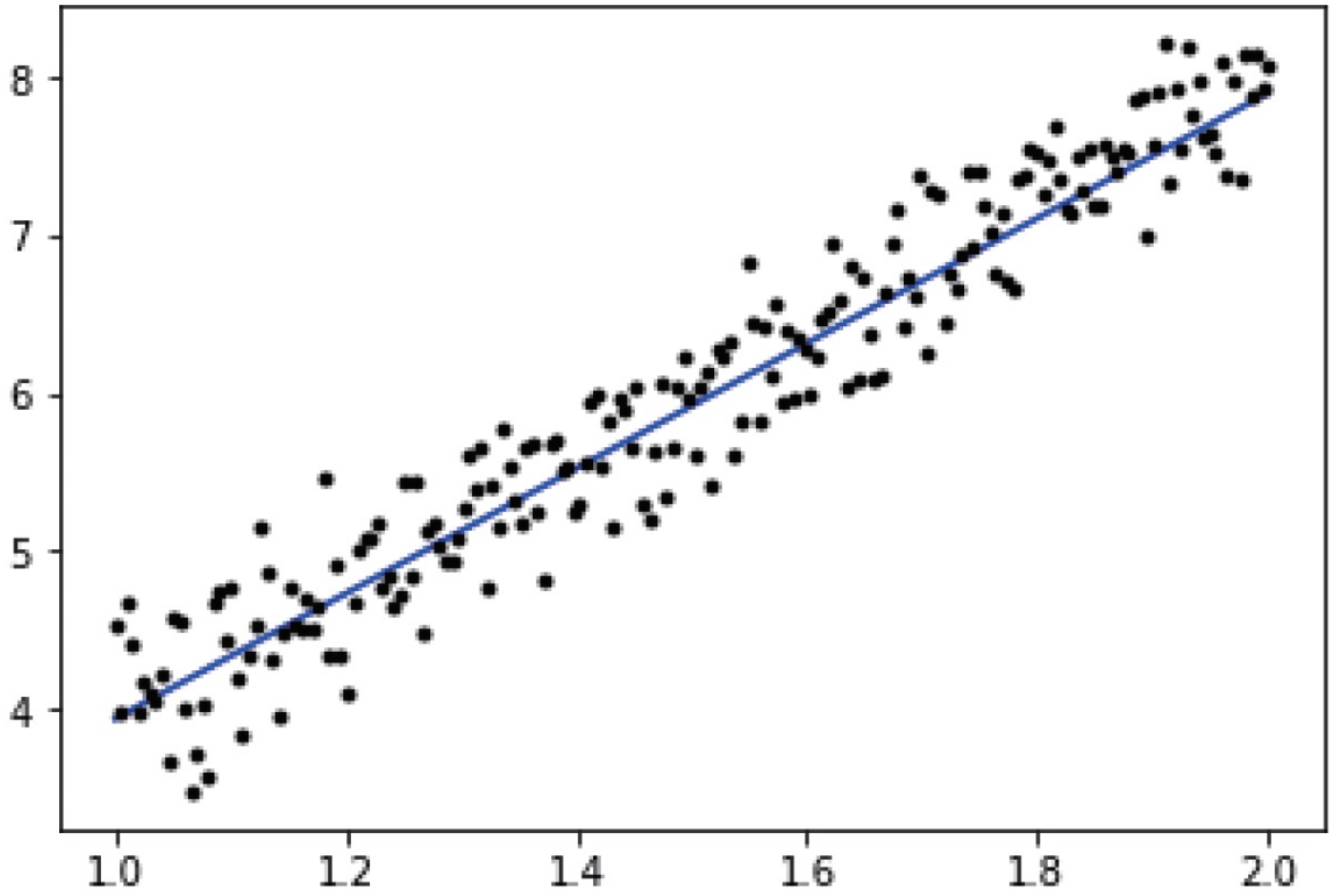
```
loss_and_metrics = model.evaluate(X, y, batch_size=128)

classes = model.predict(new_X, batch_size=128)
```

평가와 예측

Keras 예제 #1 선형 회귀

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# 가상적인 데이터 생성
X = data = np.linspace(1,2,200)          # 시작값=1, 종료값=2, 개수=200
y = x*4 + np.random.randn(200) * 0.3   # x를 4배로 하고 편차 0.3정도의 가우시안 잡음추가

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(1, input_dim=1, activation='linear'))
model.compile(optimizer='sgd', loss='mse', metrics=['mse'])
model.fit(X, y, batch_size=1, epochs=30)

predict = model.predict(data)

plt.plot(data, predict, 'b', data, y, 'k.') # 첫 번째 그래프는 파란색 마커로
plt.show()                                  # 두 번째 그래프는 검정색 .으로 그린다.
```

Keras 예제 #1 실행결과



Keras 예제 #2 XOR

```
import tensorflow as tf
import numpy as np

X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(2, input_dim=2, activation='sigmoid'))
model.add(tf.keras.layers.Dense(1,  activation='sigmoid'))

sgd = tf.keras.optimizers.SGD(lr=0.1)
model.compile(loss='mean_squared_error', optimizer=sgd)

model.fit(X, y, batch_size=1, epochs=1000)
print(model.predict(X))
```

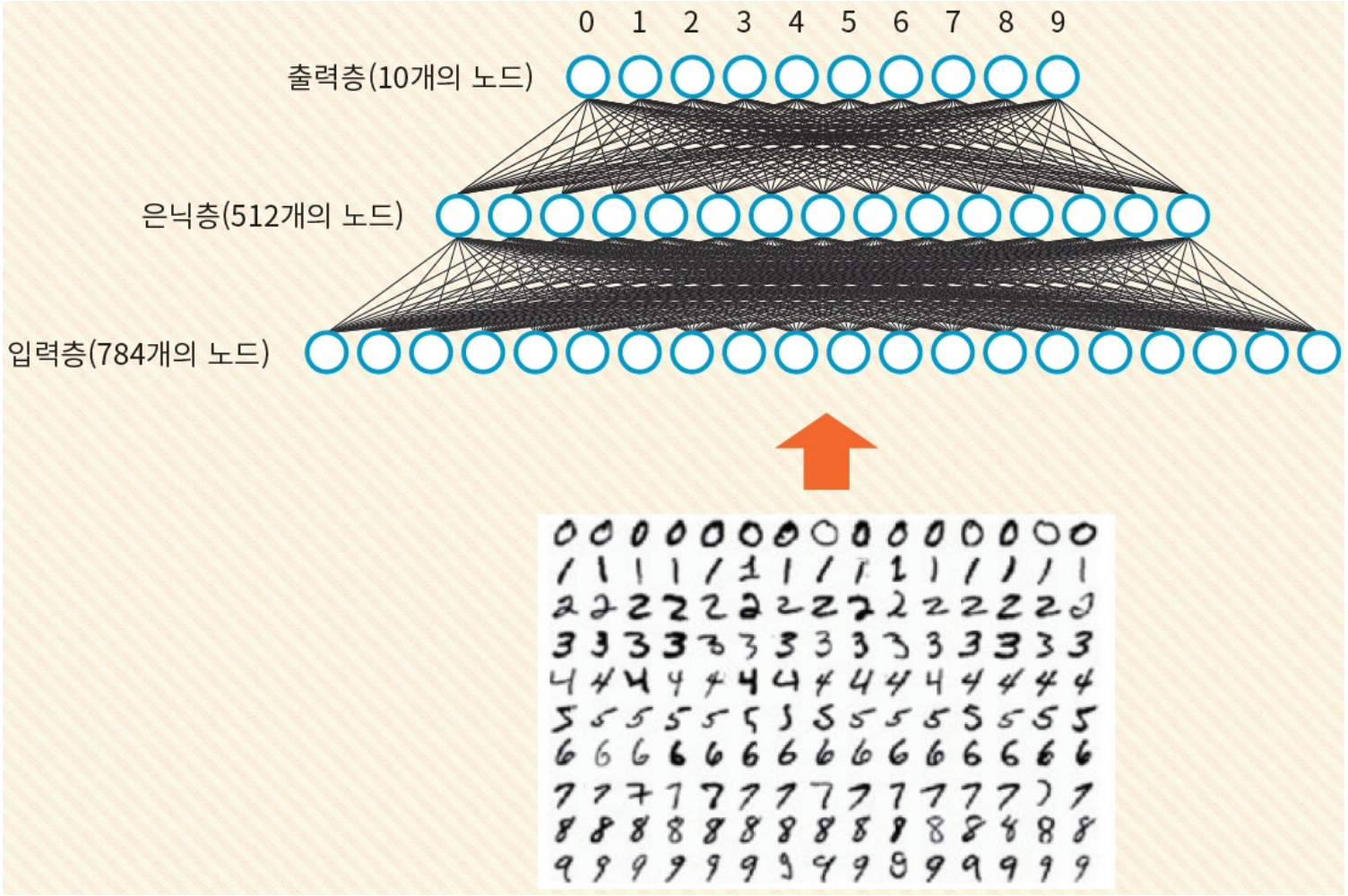
Keras 예제 #2 실행결과

```
[[0.19608304]
 [0.64313614]
 [0.6822454 ]
 [0.53627   ]]
```

```
[[0.02743807]
 [0.9702845 ]
 [0.9704155 ]
 [0.03712982]]
```

반복횟수(에포크)를 10000으로 늘린다면

Lab: MLP를 사용한 MNIST 숫자인식



Lab: MLP를 사용한 MNIST 숫자인식

```
import tensorflow as tf

batch_size = 128   # 가중치를 변경하기 전에 처리하는 샘플의 개수
num_classes = 10  # 출력 클래스의 개수
epochs = 20                    # 에포크의 개수

# 데이터를 학습 데이터와 테스트 데이터로 나눈다.
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# 입력 이미지를 2차원에서 1차원 벡터로 변경한다.
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

# 입력 이미지의 픽셀 값이 0.0에서 1.0 사이의 값이 되게 한다.
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
```

Lab: MLP를 사용한 MNIST 숫자인식

```
# 클래스의 개수에 따라서 하나의 출력 픽셀만이 1이 되게 한다.
# 예를 들면 1 0 0 0 0 0 0 0 0 0과 같다.

y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)

# 신경망의 모델을 구축한다.
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(512, activation='sigmoid', input_shape=(784,)))
model.add(tf.keras.layers.Dense(num_classes, activation='sigmoid'))

model.summary()

sgd = tf.keras.optimizers.SGD(lr=0.1)
```

Lab: MLP를 사용한 MNIST 숫자인식

```
# 손실 함수를 제곱 오차 함수로 설정하고 학습 알고리즘은 SGD 방식으로 한다.
model.compile(loss='mean_squared_error',
              optimizer=sgd,
              metrics=['accuracy'])

# 학습을 수행한다.
history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs)

# 학습을 평가한다.
score = model.evaluate(x_test, y_test, verbose=0)
print('테스트 손실값:', score[0])
print('테스트 정확도:', score[1])
```

실행결과

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 512)               401920
_________________________________________________________________
dense_5 (Dense)              (None, 10)                5130
=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
...
Epoch 18/20
60000/60000 [==============================] - 1s 18us/sample - loss: 0.0344 -
acc: 0.8480
Epoch 19/20
60000/60000 [==============================] - 1s 18us/sample - loss: 0.0335 -
acc: 0.8505
Epoch 20/20
60000/60000 [==============================] - 1s 18us/sample - loss: 0.0328 -
acc: 0.8530
테스트 손실값: 0.031483046317100 53
테스트 정확도: 0.8628
```

Summary

- 입력층과 출력층 사이에 은닉층(hidden layer)을 갖은 신경망을 MLP임
- MLP를 학습시키기 위하여 역전파 알고리즘이 이용
- 역전파 알고리즘은 신경망 학습 알고리즘의 근간임
  1. 입력이 주어지면 순방향으로 계산하여 출력을 계산
  2. 실제 출력과 기대 출력 간의 오차를 계산
  3. 오차를 역방향으로 전파하면서 오차를 줄이는 방향으로 가중치를 변경