In [1]:

```
import numpy as np
```

In [2]:

```
import numpy as np

def mean_squared_error(y, t):
    return np.sum((y - t) ** 2) / len(y)

t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
y1 = [0.1, 0.0, 0.6, 0.0, 0.05, 0.1, 0.05, 0.1, 0.0, 0.0]
y2 = [0.1, 0.1, 0.05, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]

print("예측이 y1일때 MSE : ", round(mean_squared_error(np.array(y1), np.array(t)), 4))
print("예측이 y2일때 MSE : ", round(mean_squared_error(np.array(y2), np.array(t)), 4))
```

```
예측이 y1일때 MSE :  0.0195
예측이 y2일때 MSE :  0.1295
```

In [3]:

```
def cross_entropy_error(y, t):
    delta = 1e-7
    return -np.sum(t * np.log(y + delta))

t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
y1 = [0.1, 0.0, 0.6, 0.0, 0.05, 0.1, 0.05, 0.1, 0.0, 0.0]
y2 = [0.1, 0.1, 0.05, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]

print("예측이 y1일때 CEE : ", round(cross_entropy_error(np.array(y1), np.array(t)), 4))
print("예측이 y2일때 CEE : ", round(cross_entropy_error(np.array(y2), np.array(t)), 4))
```

```
예측이 y1일때 CEE :  0.5108
예측이 y2일때 CEE :  2.9957
```

In [4]:

```
# 미니 배치용 CEE
def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)

    batch_size = y.shape[0]

    return -np.sum(t * np.log(y)) / batch_size
```

In [6]:

```
import numpy as np
from dataset.mnist import load_mnist

(x_train, t_train), (x_test, t_test) = load_mnist(normalize = True, one_hot_label = True)

print("x_train 형성 : ", x_train.shape)
print("y_train 형성 : ", t_train.shape)

train_size = x_train.shape[0]
batch_size = 10
batch_mask = np.random.choice(train_size, batch_size)

x_batch = x_train[batch_mask]
t_batch = t_train[batch_mask]

print("train data 크기에서 Mini-Batch 수만큼 랜덤하게 추출한 값 : ", batch_mask, sep = "₩n")
print("Mini-Batch 처리된 x_train : ", x_batch, sep = "₩n")
print("Mini-Batch 처리된 t_train : ", t_batch, sep = "₩n")
```

```
x_train 형성 :  (60000, 784)
y_train 형성 :  (60000, 10)
train data 크기에서 Mini-Batch 수만큼 랜덤하게 추출한 값 :
[58010   814 38728 16723 19723 38011 42556 33225 34798 21197]
Mini-Batch 처리된 x_train :
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
Mini-Batch 처리된 t_train :
[[0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]]
```

In [7]:

```
def numerical_diff(f, x):
    h = 1e-4
    return ( f(x + h) - f(x - h) ) / (2 * h)

def function_1(x):
    return 0.1 * x ** 2 + 0.1 * x

print("x = 5에서의 수치 미분 결과 : ", numerical_diff(function_1, 5))
print("x = 10에서의 수치 미분 결과 : ", numerical_diff(function_1, 10))
```

```
x = 5에서의 수치 미분 결과 :  1.0999999999983245
x = 10에서의 수치 미분 결과 :  2.099999999991553
```

In [8]:

```
def partial_diff(f, x):
    h = 1e-4
    return ( f(x + h) - f(x - h) ) / (2 * h)

def function_1(x0):
    x1 = 4
    return 3 * (x0 ** 4) + 2 * (x0 ** 2) * (x1 ** 2) + 7 * (x1 ** 4)

print("x0 = 3, x1 = 4에서의 x0에 대한 편미분 결과 : ", partial_diff(function_1, 3))
```

```
x0 = 3, x1 = 4에서의 x0에 대한 편미분 결과 :  516.0000003616005
```

In [10]:

```
def partial_diff(f, x):
    h = 1e-4
    return ( f(x + h) - f(x - h) ) / (2 * h)

def function_2(x1):
    x0 = 4
    return 3 * (x0 ** 4) + 2 * (x0 ** 2) * (x1 ** 2) + 7 * (x1 ** 4)

print("x0 = 3, x1 = 4에서의 x1에 대한 편미분 결과 : ", partial_diff(function_2, 4))
```

```
x0 = 3, x1 = 4에서의 x1에 대한 편미분 결과 :  2048.0000011207267
```

In [12]:

```
def numerical_gradient(f, x):
    h = 1e-4
    grad = np.zeros_like(x)

    for idx in range(x.size):
        tmp_val = x[idx]

        # f(x + h) 계산
        x[idx] = tmp_val + h
        fxh1 = f(x)

        # f(x - h) 계산
        x[idx] = tmp_val - h
        fxh2 = f(x)

        grad[idx] = (fxh1 - fxh2) / (2 * h)
        # 값 복원
        x[idx] = tmp_val

    return grad

def function_2(x):
    return 3 * (x[0] ** 4) + 2 * (x[0] ** 2) * (x[1] ** 2) + 7 * (x[1] ** 4)

print("x0 = 3, x1 = 4에서의 기울기 : ", numerical_gradient(function_2, np.array([3.0, 4.0])))
```

```
x0 = 3, x1 = 4에서의 기울기 :  [ 516.00000036 1936.00000112]
```

In [13]:

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

class Sigmoid:
    def __init__(self):
        self.out = None

    def forward(self, x):
        out = sigmoid(x)
        self.out = out
        return out

    def backward(self, dout):
        dx = dout * (1.0 - self.out) * self.out
        return dx
```