4장. 신경망 학습

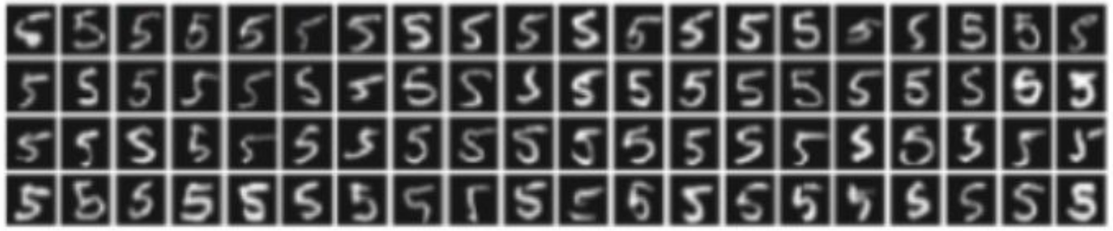
최혜원

4장 학습 목표

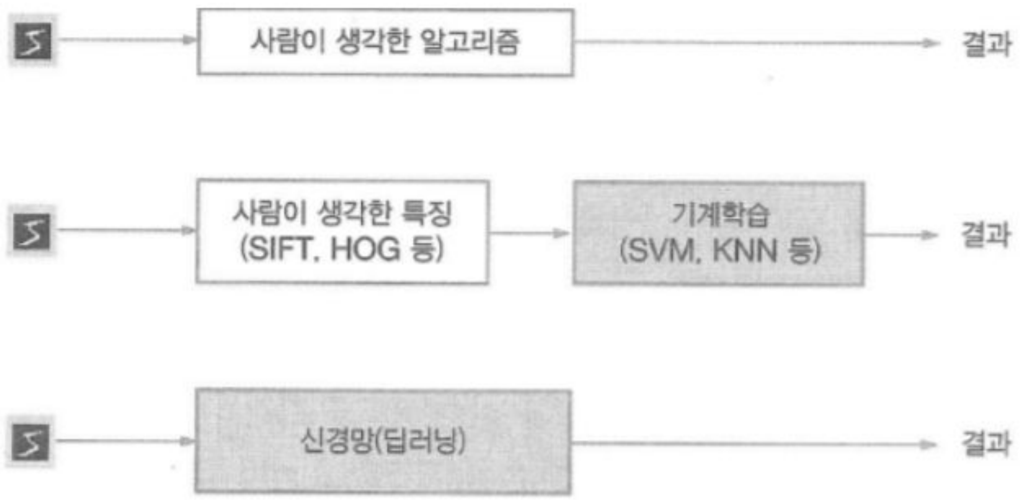
- 손실함수의 값을 가급적 작게 만드는 기법으로, 함수의 기울기를 활용하는 경사법 소개.
- 학습 : 훈련 데이터로부터 가중치 매개변수의 최적 값을 자동으로 획득하는 것.
- 손실함수 : 신경망이 학습할 수 있도록 해주는 지표.

4.1 데이터에서 학습한다

- 4. 1. 1. 데이터 주도 학습
  - *Ex> MNIST 손글씨 숫자를 학습*



손글씨 숫자 '5'의 예: 사람마다 자신만의 필체가 있다.



4.1 데이터에서 학습한다

- 4. 1. 2. 훈련데이터와 시험데이터
  - 문제를 범용적으로 다루기 위해서 train data 와 test data로 나눈다.
  - Overfitting 을 피하는 것이 중요하다.

4.2 손실함수

- 손실함수(loss function) : 신경망 성능을 나타내는 지표
- 일반적으로 평균 제곱 오차와 교차 엔트로피 오차를 사용

4.2 손실함수

- 4. 2. 1 평균 제곱 오차 (mean squared error, MSE)

$$E = \frac{1}{2}\sum_{k}(y_k - t_k)^2$$

Yk : 신경망의 출력, tk: 정답 레이블, k : 차원 수

- 예시:

```
y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
```

Softmax

정답 레이블

- 코드:

```
def mean_squared_error(y, t):
    return 0.5 * np.sum((y - t)**2)
```

4.2 손실함수

- 4. 2. 1 평균 제곱 오차 (mean squared error, MSE)
- 예시:

```python
import numpy as np
def mean_squared_error(y, t):
    return 0.5 * np.sum((y - t)**2)

t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0] # 정답은 2
y1 = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0] #2일 확률이 가장 높다고 추정(0.6)

print( mean_squared_error(np.array(y1), np.array(t)) )


y2 = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0] #7일 확률이 가장 높다고 추정(0.6)
print( mean_squared_error(np.array(y2), np.array(t)) )

```

```
0.09750000000000003
0.5975
```

=> 값이 작을 수록 정답에 더 가깝다

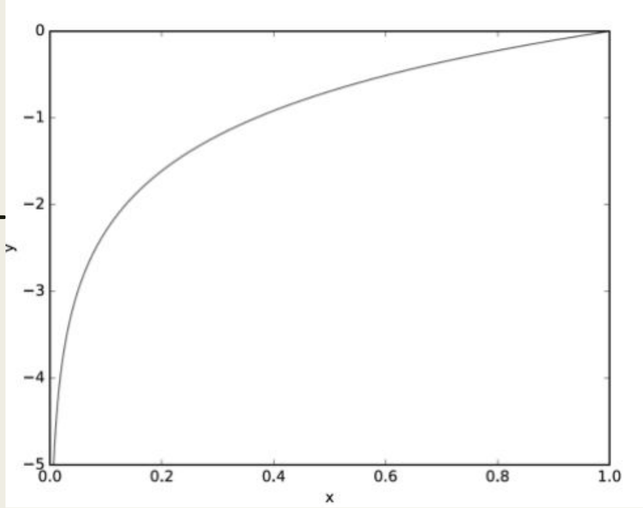
4.2 손실함수

- 4. 2. 2 교차 엔트로피 오차 (cross entropy error, CEE)

$$E = -\sum_{k} t_k \log y_k$$

Yk : 신경망의 출력, tk: 정답 레이블, k : 차원 수, log의 밑은 e

- Tk가 정답일 때만 값이 1이고
  나머지는 0이기 때문에 실질적으로
  정답일 때의 추정yk의 자연로그를 계산



-log0.6 = 0.51

-log0.1 = 2.30

4.2 손실함수

- 4. 2. 2 교차 엔트로피 오차 (cross entropy error, CEE)
- 코드 :

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

t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]                              # 정답은 2
y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0] # 신경망이 2로 추정

print(cross_entropy_error(np.array(y), np.array(t)))

y = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0] # 신경망이 7로 추정

print(cross_entropy_error(np.array(y), np.array(t)))
```

```
0.510825457099338
2.302584092994546
```

Delta : log() 함수 안에 0이 입력되면 마이너스 무한대의 값이 리턴됨.

4.2 손실함수

$$E = -\frac{1}{N}\sum_{n}\sum_{k} t_{nk} \log y_{nk}$$

- 4. 2. 3 미니배치 학습
- 훈련데이이터가 N개->평균을 낸다 : 평균 손실 함수
- 그러나, 훈련데이터 일일이 손실함수 평균을 계산하는 것은 시간이 오래 걸림.
  -> 일부(미니 배치)만 골라 학습 수행 : **미니배치 학습**

```
train_size = x_train.shape[0] #60000
batch_size = 10
batch_mask = np.random.choice(train_size, batch_size)

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

```
np.random.choice(60000, 10)
```

```
array([23594, 21588, 46387, 57523, 44773, 13334, 10545, 20792, 37820,
       17252])
```

4.2 손실함수

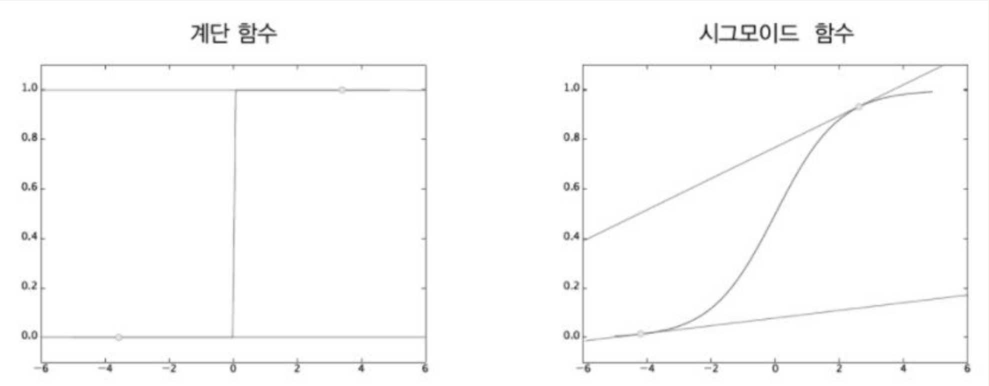
- 4. 2. 4 (배치용) 교차 엔트로피 오차 구현

```
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]        # 배치 사이즈를 의미

    return -np.sum( np.log( y[np.arange(batch_size), t] )) / batch_size # 원핫인
코딩이 아닐경우
```

y[np.arrange(batch_size), t] : 각 정답 레이블에 해당하는 신경망의 출력 추출
Ex)
Batch_size = 5이면,np.arrange(batch_size) = [0, 1, 2, 3, 4]
T = [2, 7, 0, 9, 4]이런식으로 one-hot 인코딩이 안되어서 적용되어있다고 가정,

4.2 손실함수

- 4. 2. 5 왜 손실함수를 설정하는가?
- 정확도가 아닌 손실함수의 값을 택하는 이유
- 정확도는 값이 불연속적으로 변화함.
  Ex) 매개변수가 약간 변하면 값에
  변동이 거의 없음.
  값을 맞췄냐, 안맞췄냐에 기준을
  두기 때문에 불연속적으로 값 변화



- 계단함수, 시그모이드의 차이
- 신경망학습은 손실함수의 미분을 계산하고 미분값이 0이 되는 쪽으로 매개변수를 갱신해준다.

4.3 수치 미분(Numerical differentiation)

- 4. 3. 1 미분

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- X의 작은 변화가 함수 f(x)를 얼마나 변화시키느냐를 의미.
- 구현 :

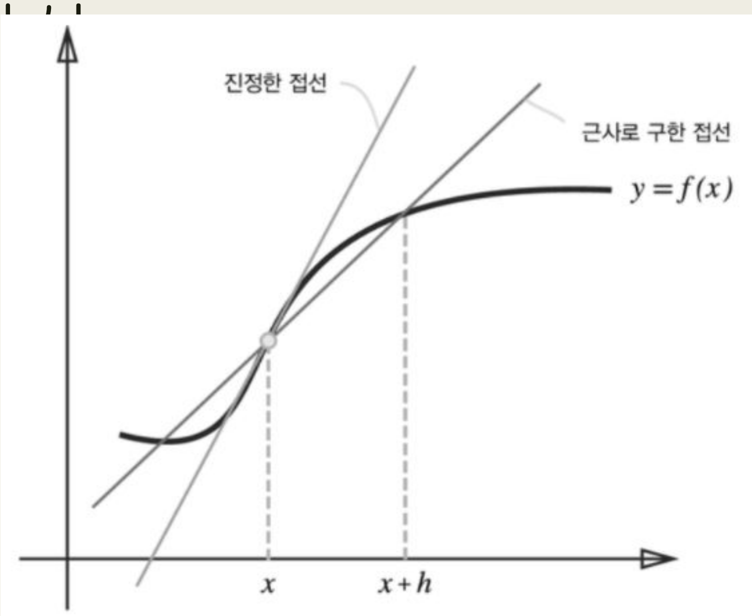
```
# 나쁜 구현 예
def numerical_diff(f, x):
    h = 10e - 50

    return (f(x + h) - f(x)) / h
```

4.3 수치 미분(Numerical differentiation)

- 4. 3. 1 미분

```
# 나쁜 구현 예
def numerical_diff(f, x):
    h = 10e - 50

    return (f(x + h) - f(x)) / h
```

- 개선할 점 1: h값 너무 작은 값 -> 10^(-4)로 대체
- 개선할 점 2: 오차 줄이기 -> 중심차분을 계



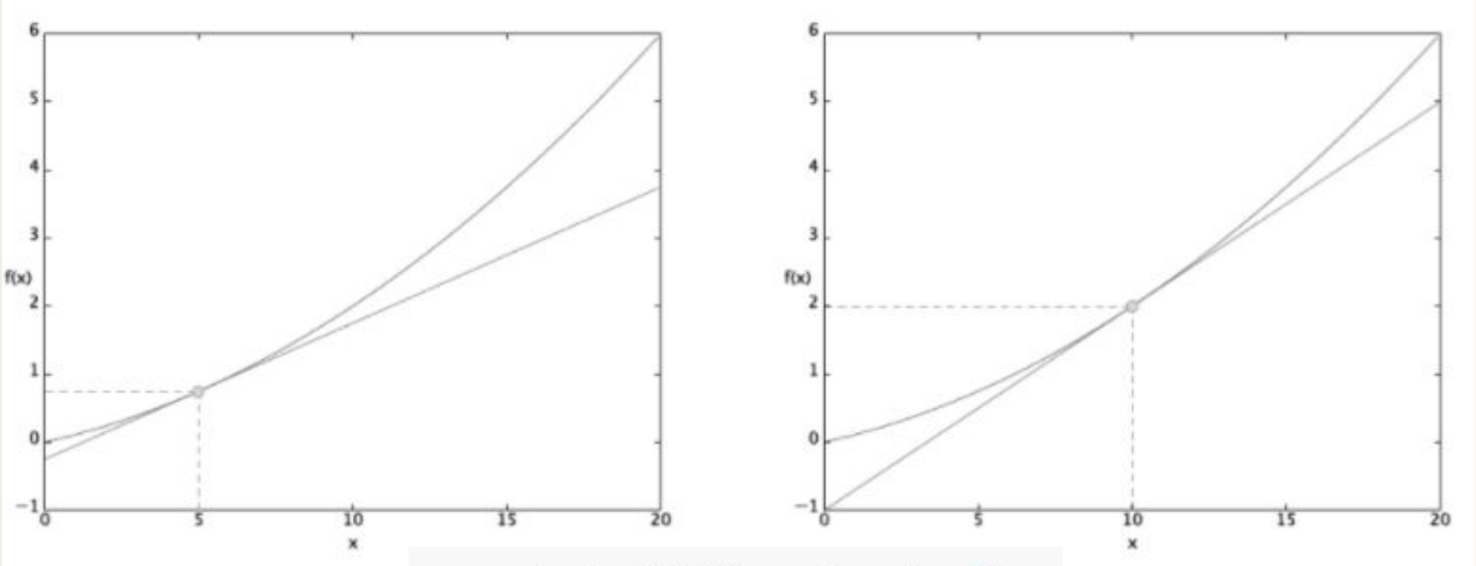
- 개선 식

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

4.3 수치 미분(Numerical differentiation)

- 4. 3. 2 수치 미분의 예

식 1 $$y = 0.01x^2 + 0.1x$$




```
numerical_diff(function_1, 5)
>>> 0.19999999

numerical_diff(function_1 ,10)
>>> 0.29999999
```

4.3 수치 미분(Numerical differentiation)

- 4. 3. 3 편미분
- 편미분 : 변수가 여럿인 함수에 대한 미분

식 2 $$f(x_0, x_1) = x_0^2 + x_1^2$$

- 예시: x0 = 3, x1 = 4일 때 $\frac{\partial f}{\partial x_0}$ 구하기

```
def function_tmp1(x0):
    return x0 * x0 + 4.0 ** 2.0

numerical_diff(function_tmp1, 3.0)
```

4.3 수치 미분(Numerical differentiation)

- 4. 3. 4 기울기
  이전 에는 x0과 x1의 편미분을 변수별로 따로 계산.
- X0과 x1의 편미분을 동시에 계산하는 방법? (기울기를 계산)

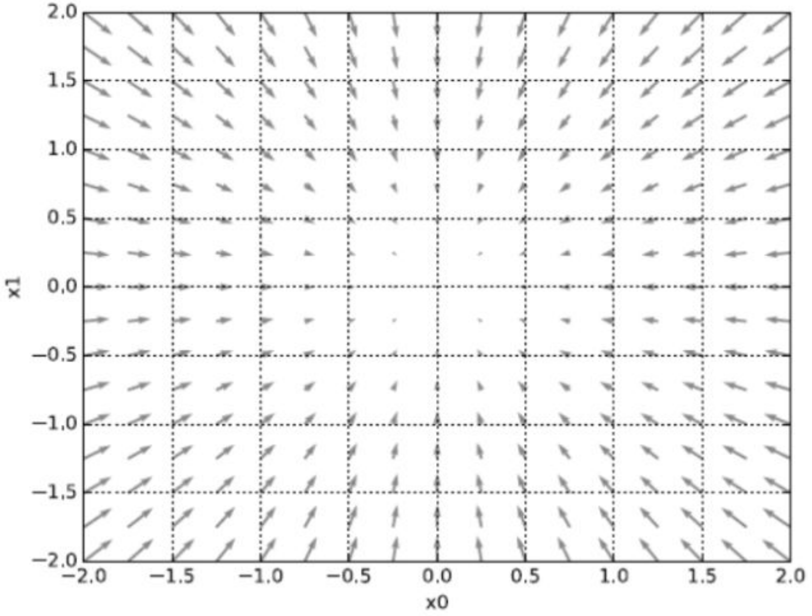
```python
def numerical_gradient(f, x):
    h = 1e-4                    # 0.0001
    grad = np.zeros_like(x)

    for idx in range(x.size): # x의 요소수만큼 반복
        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)  # 중심차분
        x[idx] = tmp_val                   # 값 복원

    return grad
```

```python
print(numerical_gradient(function_2, np.array([3.0, 4.0])))

print(numerical_gradient(function_2, np.array([0.0, 2.0])))

print(numerical_gradient(function_2, np.array([3.0, 0.0])))

```

```
[6. 8.]
[0. 4.]
[6. 0.]
```

4.3 수치 미분(Numerical differentiation)

- 4. 3. 4 기울기
  기울기의 의미 :

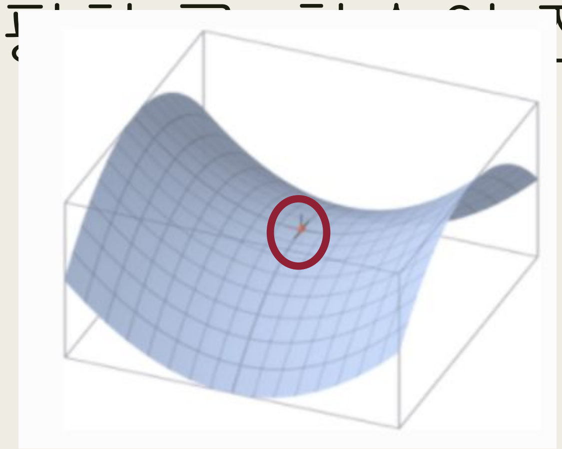


$f(x_0, x_1) = x_0^2 + x_1^2$의 기울기

기울기가 가리키는 쪽은 각 장소에서 함수의 출력 값을 가장 크게 줄이는 방향

4.4 기울기

- 4. 4. 1 경사법(경사 하강법)
  : 기울기를 잘 이용해 함수의 최솟값, 가능한 한 작은 값을 찾으려는 것.
- 극솟값 : 국소적인 최솟값. 한정된 범위에서의 최솟값인 점.
- 안장점(saddle point) : 기울기가 0. (말의 안장)
- 고원(plateau) : 평 [illegible] 진행되지 않는 정체기



4.4 기울기

- 4. 4. 1 경사법(경사 하강법)
  - 현 위치에서 기울어진 방향으로 일정 거리만큼 이동
  - 다음 이동한 곳에서도 마찬가지로 기울기를 구함
  - 또 기울어진 방향으로 나아감
  - 함수의 값을 점점 줄여나감
- 수식 :

$$x_0 = x_0 - \eta \frac{\partial f}{\partial x_0}$$

$$x_1 = x_1 - \eta \frac{\partial f}{\partial x_1}$$

기호 η (eta) : 갱신하는 양 : 학습률(learning rate)
: 한 번의 학습으로 얼마만큼 학습해야 할 지,
매개변수 값을 얼마나 갱신할 지를 정하는 것

- 위 예시는 변수 두 개, 1회의 해당하는 갱신
- 학습률의 값은 미리 특정값으로 정해두어야 한다.

4.4 기울기

- 4. 4. 1 경사법(경사 하강법)
- 구현 :

```python
def gradient_descent(f, init_x, lr=0.01, step_num=100):
    x = init_x

    for i in range(step_num):
        grad = numerical_gradient(f, x)#함수의 기울기
        x -= lr * grad

    return x
```

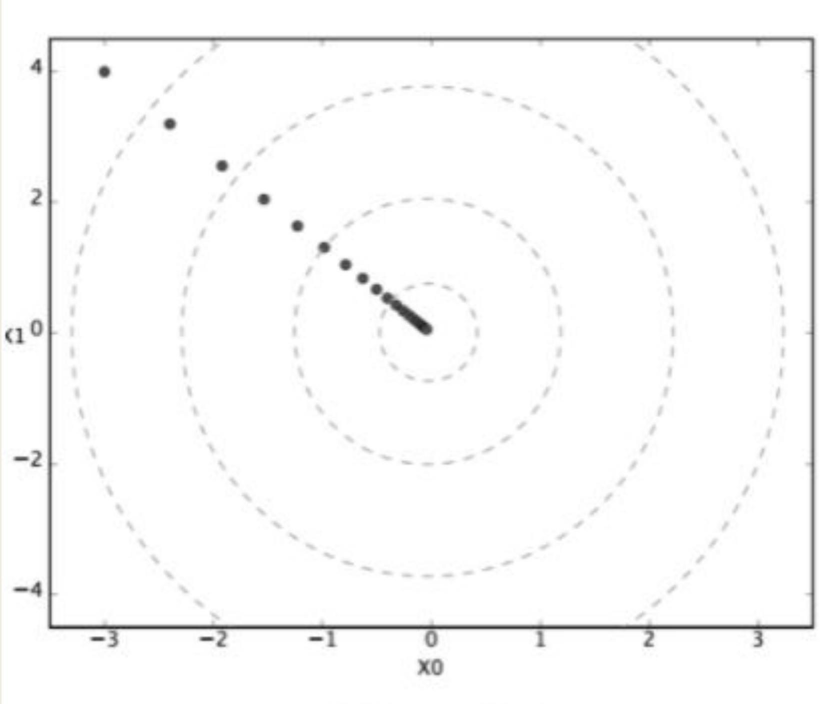
F : 최적화하려는 함수, init_x :초깃값, lr: 학습률, step_num: 반복 횟수

4.4 기울기

- 4. 4. 1 경사법(경사 하강법)
- 응용 :

```python
def function_2(x):
    return x[0]**2 + x[1]**2

init_x = np.array([-3.0, 4.0])

gradient_descent(function_2, init_x=init_x, lr=0.1, step_num=100)

```

```
array([-6.11110793e-10,  8.14814391e-10])
```



경사법에 의한 갱신 과정

4.4 기울기

- 4. 4. 1 경사법(경사 하강법)
- 응용 : 학습률에 따른 실험

```
init_x = np.array([-3.0, 4.0])
gradient_descent(function_2, init_x=init_x, lr=10.0, step_num=100)
```

```
array([-2.58983747e+13, -1.29524862e+12])
```

```
init_x = np.array([-3.0, 4.0])
gradient_descent(function_2, init_x=init_x, lr=1e-10, step_num=100)
```

```
array([-2.99999994,  3.99999992])
```

하이퍼 파라미터

4.4 기울기

- 4. 4. 2 신경망에서의 기울기
- 가중치 매개변수에 대한 손실함수의 기울
- W의 shape == 손실함수의 기울기 shape

식 1.

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{pmatrix}$$

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{31}} \\ \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{32}} \end{pmatrix}$$

```python
def softmax(a):
    exp_a = np.exp(a)
    sum_exp_a = np.sum(exp_a)
    y = exp_a / sum_exp_a

    return y
class simpleNet:
    def __init__(self):
        self.W = np.random.randn(2,3)

    def predict(self, x):
        return np.dot(x, self.W)# x와 self.W 내적

    def loss(self, x, t):# x는 입력, t는 정답 레이블
        z = self.predict(x)
        y = softmax(z)
        loss = cross_entropy_error(y, t)#교차 엔트로피 오차 이용
        return loss
```

```python
net = simpleNet()
print(net.W) # 가중치 매개변수

x = np.array([0.6, 0.9])

p = net.predict(x)
print(p)
print( np.argmax(p) )           # 최댓값의 인덱스
t = np.array([0, 0, 1])         # 정답 레이블
print(net.loss(x, t))           # 손실 함수 구하기
```

```
[[-1.17974361 -1.15579862 -0.16282616]
 [ 0.58458667 -0.43374132  1.59271935]]
[-0.18171817 -1.08384636  1.33575172]
2
0.26866983883179174
```

4.4 기울기

- 4. 4. 2 신경망에서의 기울기
- 기울기

```
def f(W):
    return net.loss(x, t)

dW = numerical_gradient(f, net.W)

print(dW)
>>> [[0.22 0.14 -0.36]
     [0.32 0.21 -0.54]]
```

- 0.22의 의미 : w[0][0]을 h만큼 늘린다면 손실함수의 값은 0.22h만큼 증가한다는 의미 :**가중치를 감소**시켜라
- -0.54의 의미 : w[1][2] 을 h만큼 늘리면 손실함수의 값은 0.54h만큼 감소한다는 의미 :**가중치를 늘려**라
- 즉, 손실함수를 줄인다는 관점 : w12을 양의 방향으로 갱신하고, w00은 음의 방향으로 갱신해야 한다는 의미
- 값의 절대값이 클수록 갱신되는 양에 더 크게 기여함

4.5 학습 알고리즘 구현하기

- 신경망 학습의 절차 정리

1단계 – 미니배치 : 훈련데이터 중 일부를 무작위로 가져온 데이터

2단계 – 기울기 산출 : 미니배치의 손실함수 값을 줄이기 위해 각 가중치 매개변수의 기울기를 구한다.

3단계 – 매개변수 갱신 : 가중치 매개변수를 기울기 방향으로 아주 조금 갱신

4단계 – 반복 : 1~3단계를 반복.

- 미니배치를 무작위로 선정 : 확률적 경사 하강법 (SGD, stochastic gradient descent)

4.5 학습 알고리즘 구현하기

- 4. 5. 1 2층 신경망 클래스 구현하기 – init, predict

```
class TwoLayerNet:

    def __init__(self, input_size, hidden_size, output_size, weight_init_std=0.01):
        # 가중치 초기화
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)

    def predict(self, x):
        W1, W2 = self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']

        a1 = np.dot(x, W1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        y = softmax(a2)

        return y
```

4.5 학습 알고리즘 구현하기

- 4. 5. 1 2층 신경망 클래스 구현하기 – loss, numerical_gradient

```
# x : 입력 데이터, t : 정답 레이블
def loss(self, x, t):
    y = self.predict(x)

    return cross_entropy_error(y, t)

def accuracy(self, x, t):
    y = self.predict(x)
    y = np.argmax(y, axis=1)
    t = np.argmax(t, axis=1)

    accuracy = np.sum(y == t) / float(x.shape[0])
    return accuracy
```

```
# x : 입력 데이터, t : 정답 레이블
def numerical_gradient(self, x, t):
    loss_W = lambda W: self.loss(x, t)

    grads = {}
    grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
    grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
    grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
    grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

    return grads
```

4.5 학습 알고리즘 구현하기

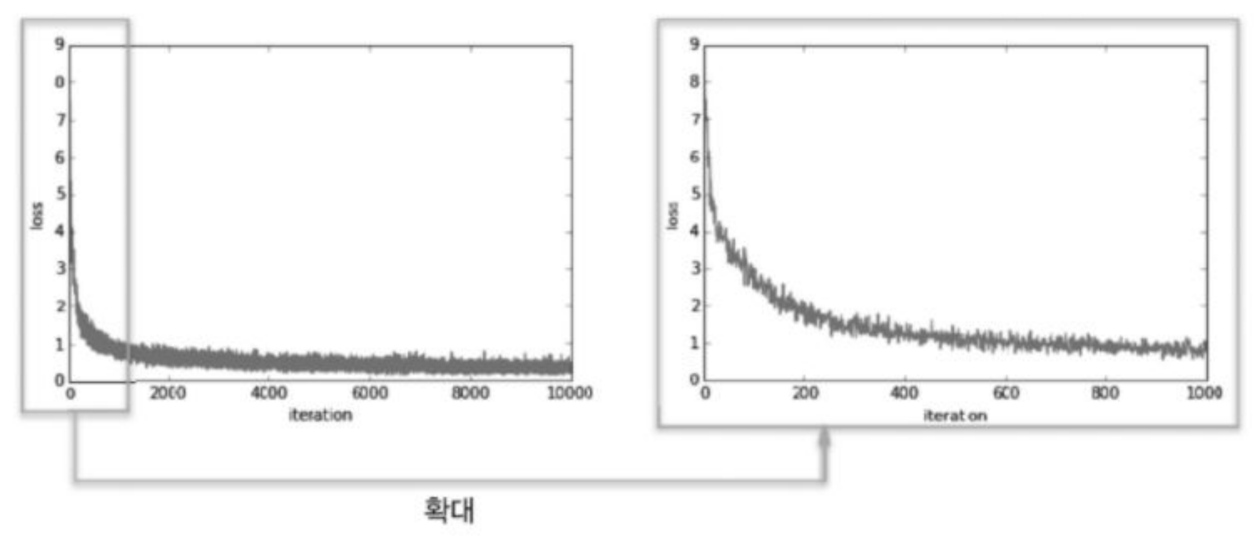
- 4. 5. 2, 3 미니배치 학습 구현하기, 시험 데이터로 평가하기

```python
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

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

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

# 하이퍼파라미터
iters_num = 10000  # 반복 횟수를 적절히 설정한다.
train_size = x_train.shape[0]
batch_size = 100   # 미니배치 크기
learning_rate = 0.1

train_loss_list = []
train_acc_list = []
test_acc_list = []

# 1에폭당 반복 수
iter_per_epoch = max(train_size / batch_size, 1)
```

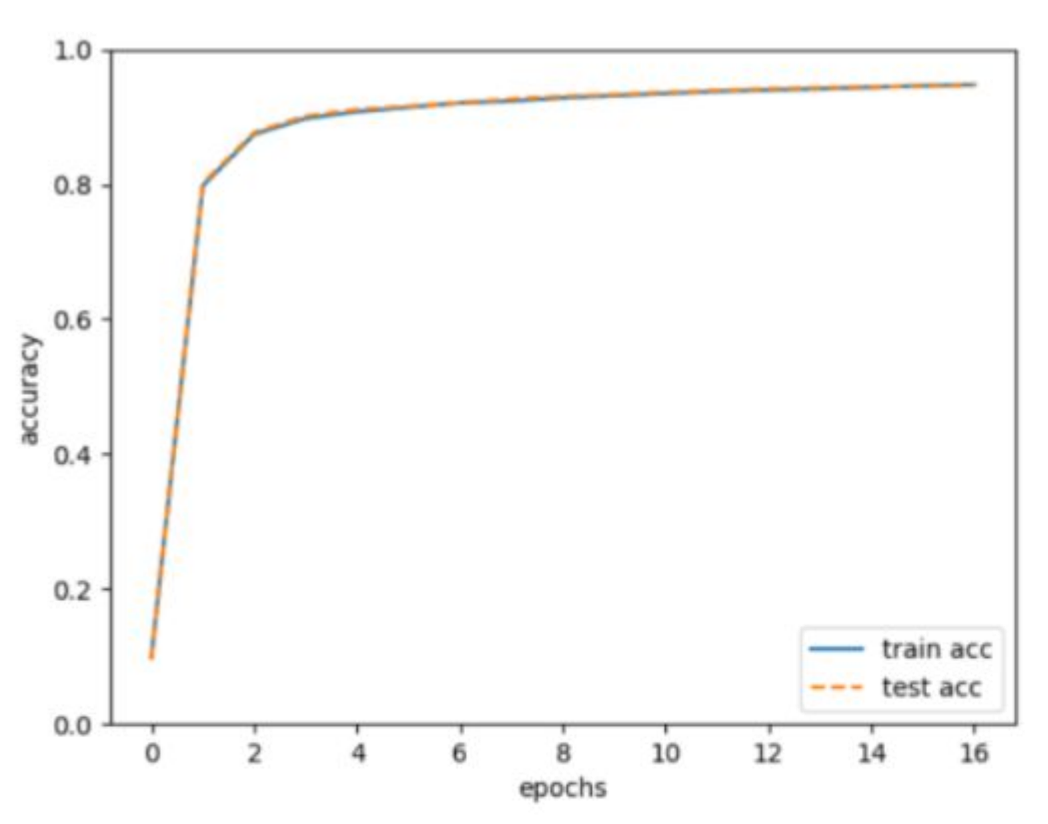
```python
for i in range(iters_num):
    # 미니배치 획득
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # 기울기 계산
    grad = network.numerical_gradient(x_batch, t_batch)
    #grad = network.gradient(x_batch, t_batch)

    # 매개변수 갱신
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    # 학습 경과 기록
    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)

    # 1에폭당 정확도 계산
    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        print("train acc, test acc | " + str(train_acc) + ", " + str(test_acc))
```

4.5 학습 알고리즘 구현하기

- 결과



손실 함수 값의 추이: 왼쪽은 10,000회 반복까지의 추이, 오른쪽은 1,000회 반복까지의 추이



훈련 데이터와 시험 데이터에 대한 정확도 추이