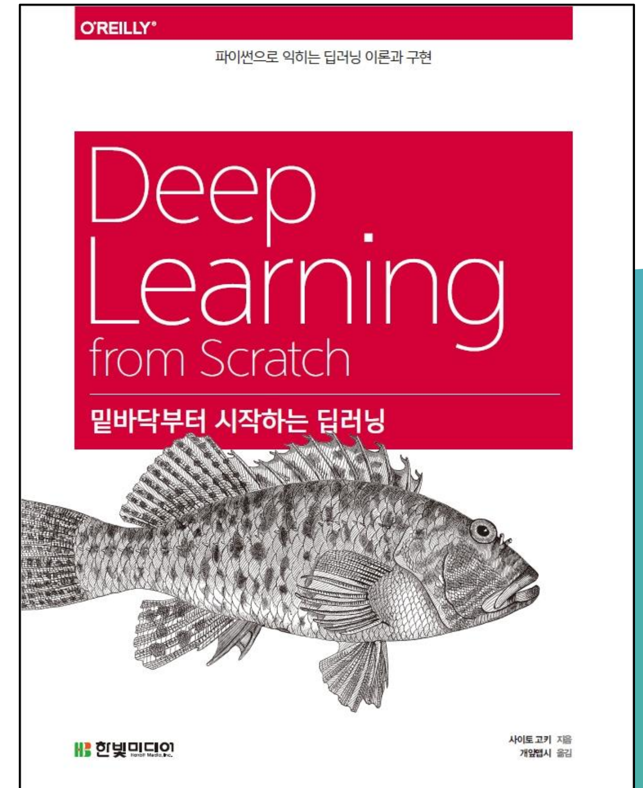
CHAPTER 6 학습 관련 기술들

밑바닥부터 시작하는 딥러닝



CHAPTER 6 학습 관련 기술들

신경망(딥러닝) 학습의 효율과 정확도를 높이기

Contents

- CHAPTER 6 학습 관련 기술들

- 6.1 매개변수 갱신
- 6.1.1 모험가 이야기
- 6.1.2 확률적 경사 하강법(SGD)
- 6.1.3 SGD의 단점
- 6.1.4 모멘텀
- 6.1.5 AdaGrad
- 6.1.6 Adam
- 6.1.7 어느 갱신 방법을 이용할 것인가?
- 6.1.8 MNIST 데이터셋으로 본 갱신 방법 비교
- 6.2 가중치의 초깃값
- 6.2.1 초깃값을 0으로 하면?
- 6.2.2 은닉층의 활성화값 분포
- 6.2.3 ReLU를 사용할 때의 가중치 초깃값
- 6.2.4 MNIST 데이터셋으로 본 가중치 초깃값 비교
- 6.3 배치 정규화
- 6.3.1 배치 정규화 알고리즘
- 6.3.2 배치 정규화의 효과
- 6.4 바른 학습을 위해
- 6.4.1 오버피팅
- 6.4.2 가중치 감소
- 6.4.3 드롭아웃
- 6.5 적절한 하이퍼파라미터 값 찾기
- 6.5.1 검증 데이터
- 6.5.2 하이퍼파라미터 최적화
- 6.5.3 하이퍼파라미터 최적화 구현하기
- 6.6 정리

6.1.2 확률적 경사 하강법(SGD)

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial L}{\partial \mathbf{W}} \qquad \text{[식 6.1]}$$

초기화 때 받는 인수인 lr은 learning rate (학습률)를 뜻한다.
이 학습률을 인스턴스 변수로 유지한다.

Update(params, grads) 메서드는 SGD 과정에서 반복해서 불린다

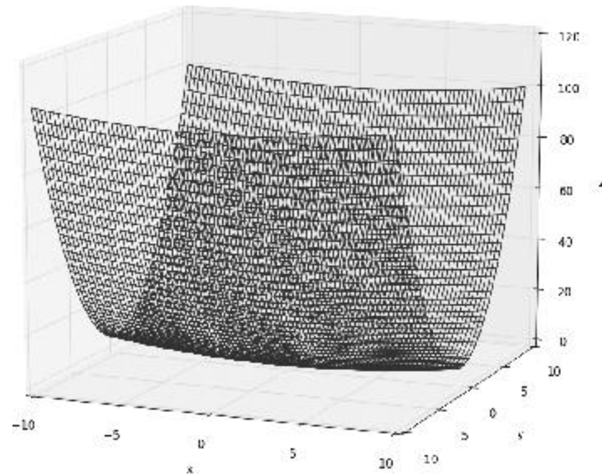
```
import numpy as np

class SGD:

    """확률적 경사 하강법 (Stochastic Gradient Descent) """

    def __init__(self, lr=0.01):
        self.lr = lr

    def update(self, params, grads):
        for key in params.keys():
            params[key] -= self.lr * grads[key]
```

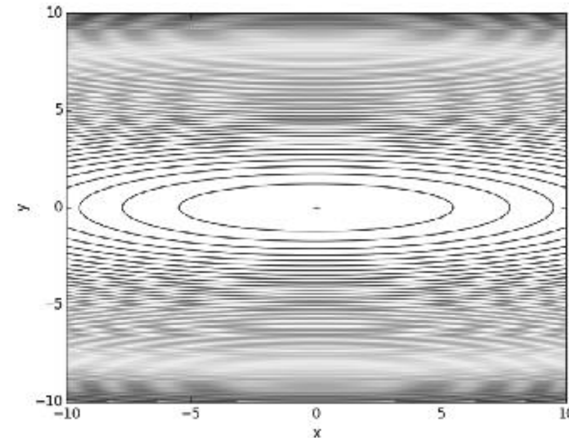
common/optimizer.py

6.1.3 SGD의 단점

$$f(x, y) = \frac{1}{20}x^2 + y^2 \qquad \text{[식 6.2]}$$

이 함수는 [그림 6-1]의 왼쪽과 같이 '밥그릇'을 x축 방향으로 늘인 듯한 모습이고, 실제로 그 등고선은 오른쪽과 같이 x축 방향으로 늘인 타원으로 되어 있다.





6.1.3 SGD의 단점

[식 6.2] 함수의 기울기를 그려보면 [그림 6-2]처럼 된다. 이 기울기는 *y*축 방향은 크고*x*축 방향은 작다는 것이 특징이다

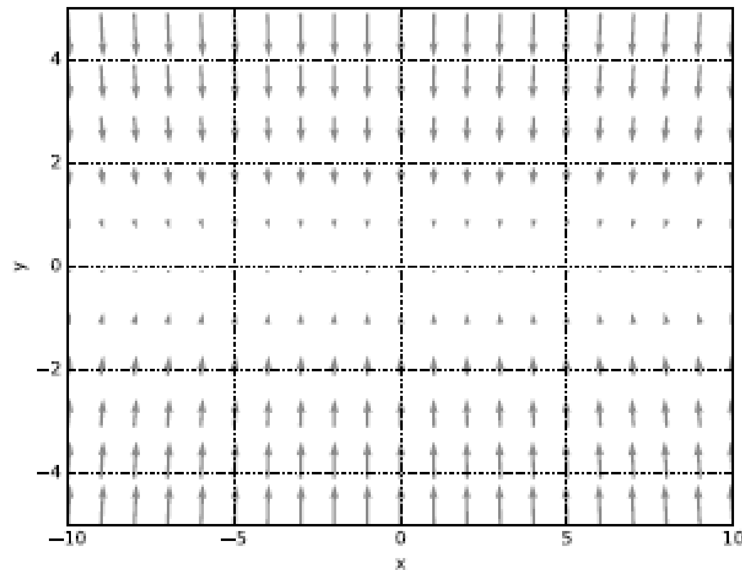


그림 6-2 $f(x, y) = \frac{1}{20}x^2 + y^2$의 기울기

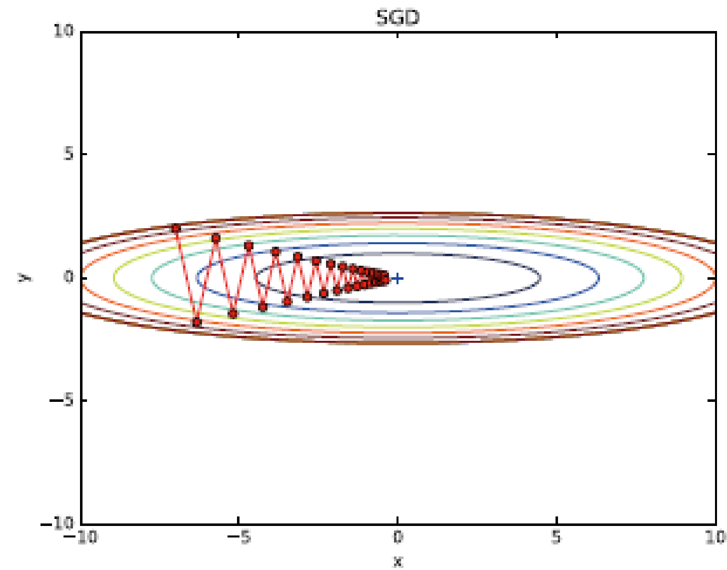


그림 6-3 SGD에 의한 최적화 갱신 경로 : 최솟값인 (0, 0)까지 지그재그로 이동하니 비효율적이다.

6.1.4 모멘텀

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}} \qquad \text{[식 6.3]}$$

$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v} \qquad \text{[식 6.4]}$$

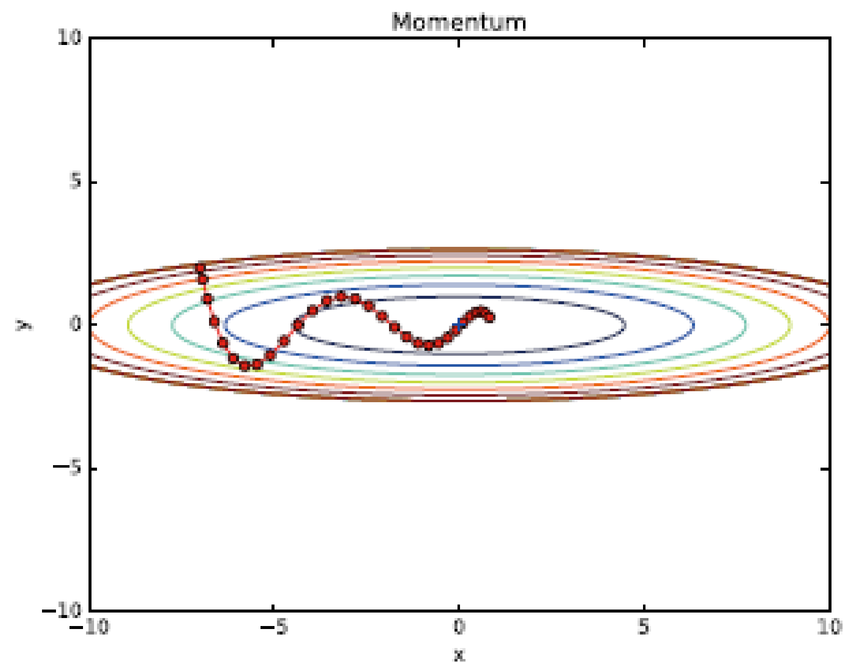
$$v(0) = -\eta \frac{\partial L}{\partial W}(0)$$
$$W(1) += v(0)$$
$$v(1) = \alpha v(0) - \eta \frac{\partial L}{\partial W}(1)$$
$$W(2) += v(1)$$

그림 6-4 모멘텀의 이미지 : 공이 그릇의 곡면(기울기)을 따라 구르듯 움직인다.

common/optimizer.py

```
class Momentum:

    """모멘텀 SGD"""

    def __init__(self, lr=0.01, momentum=0.9):
        self.lr = lr
        self.momentum = momentum
        self.v = None

    def update(self, params, grads):
        if self.v is None:
            self.v = {}
            for key, val in params.items():
                self.v[key] = np.zeros_like(val)

        for key in params.keys():
            self.v[key] = self.momentum*self.v[key] - self.lr*grads[key]
            params[key] += self.v[key]
```

그림 6-5 모멘텀에 의한 최적화 갱신 경로



또, [식 6.3]의 **αv** 항은 물체가 아무런 힘을 받지 않을 때 서서히 하강시키는 역할을 한다 (α는 0.9 등의 값으로 설정).

6.1.5 AdaGrad

신경망 학습에서는 학습률(수식에서는 η로 표기) 값이 중요
이 학습률을 정하는 효과적 기술로 학습률 감소learning rate decay가 있다

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}} \quad [식 6.5]$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}} \quad [식 6.6]$$

```python
class AdaGrad:

    """AdaGrad"""

    def __init__(self, lr=0.01):
        self.lr = lr
        self.h = None

    def update(self, params, grads):
        if self.h is None:
            self.h = {}
            for key, val in params.items():
                self.h[key] = np.zeros_like(val)

        for key in params.keys():
            self.h[key] += grads[key] * grads[key]
            params[key] -= self.lr * grads[key] / (np.sqrt(self.h[key]) + 1e-7)
```

common/optimizer.py

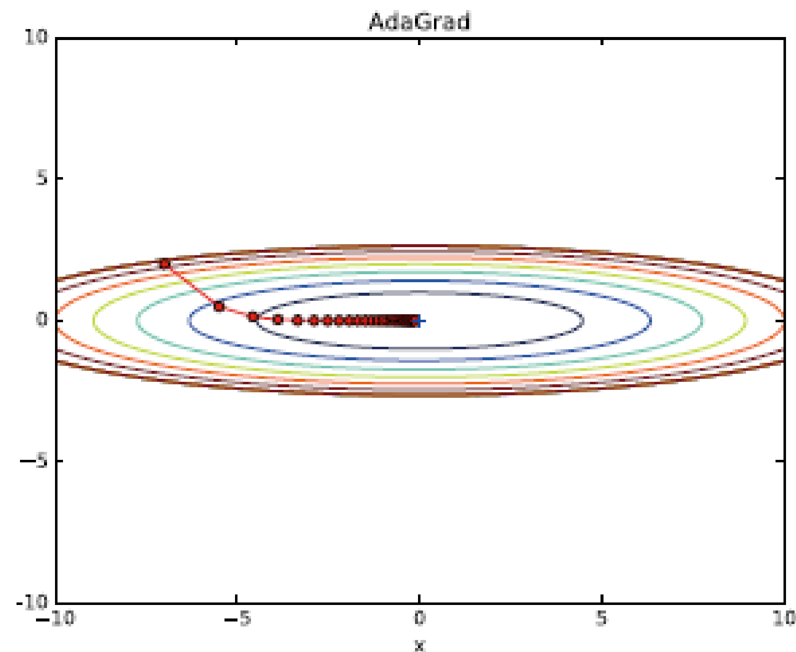
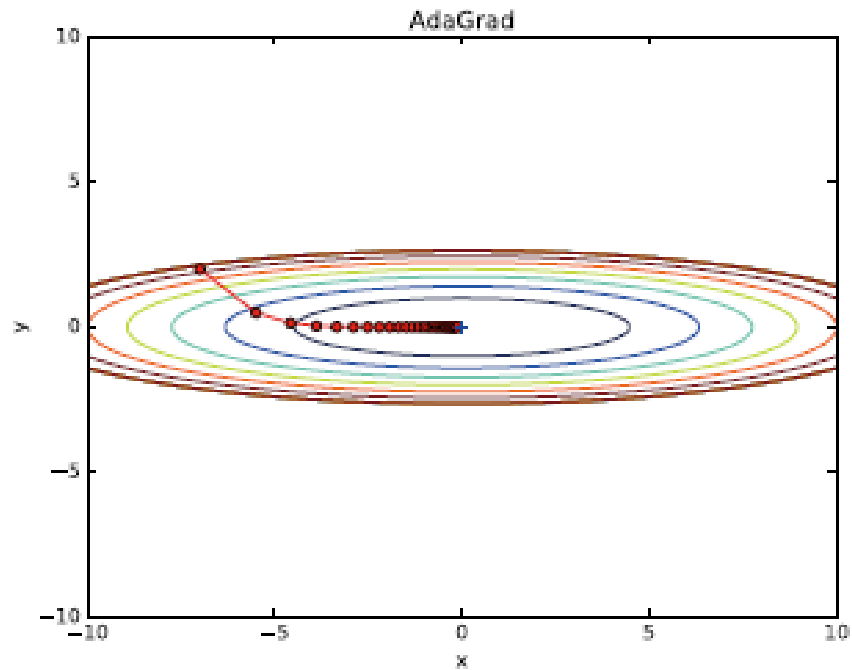


그림 6-6 AdaGrad에 의한 최적화 갱신 경로

6.1.6 Adam

AdaGrad는 매개변수의 원소마다 적응적으로 갱신 정도를 조정했다.
이런 생각에서 출발한 기법이 바로 Adam 이다.*

그림 6-6 AdaGrad에 의한 최적화 갱신 경로



common/optimizer.py

6.1.7 어느 갱신 방법을 이용할 것인가?

```python
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
import matplotlib.pyplot as plt
from collections import OrderedDict
from common.optimizer import *


def f(x, y):
    return x**2 / 20.0 + y**2


def df(x, y):
    return x / 10.0, 2.0*y

init_pos = (-7.0, 2.0)
params = {}
params['x'], params['y'] = init_pos[0], init_pos[1]
grads = {}
grads['x'], grads['y'] = 0, 0


optimizers = OrderedDict()
optimizers["SGD"] = SGD(lr=0.95)
optimizers["Momentum"] = Momentum(lr=0.1)
optimizers["AdaGrad"] = AdaGrad(lr=1.5)
optimizers["Adam"] = Adam(lr=0.3)

idx = 1
```

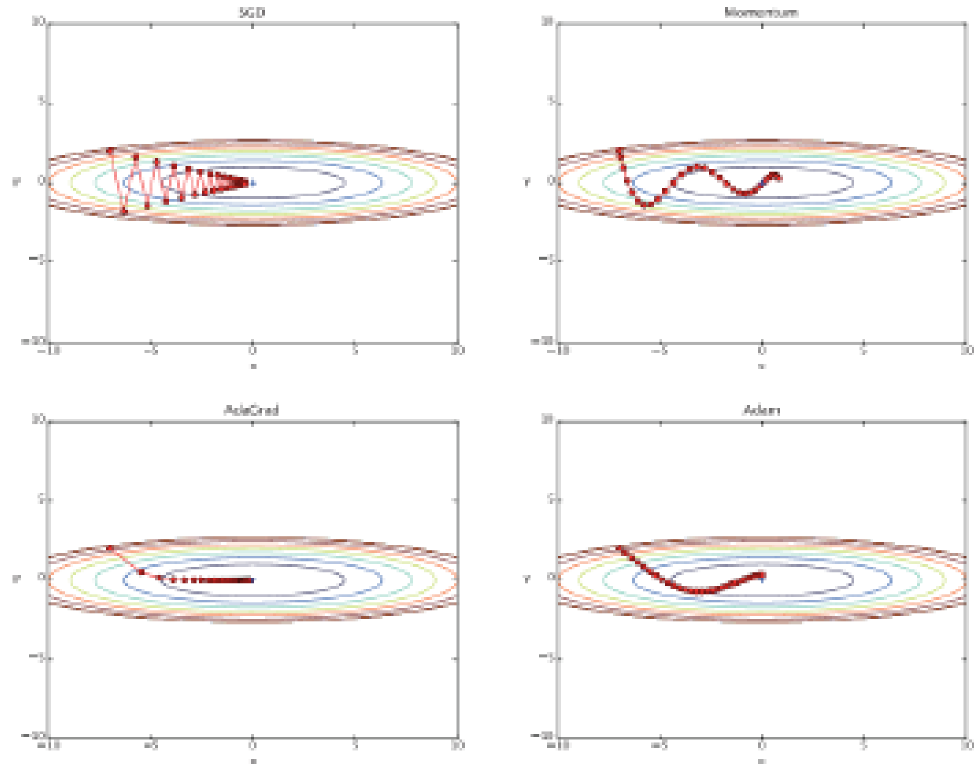
ch06/optimizer_compare_naive.py

```python
for key in optimizers:
    optimizer = optimizers[key]
    x_history = []
    y_history = []
    params['x'], params['y'] = init_pos[0], init_pos[1]

    for i in range(30):
        x_history.append(params['x'])
        y_history.append(params['y'])

        grads['x'], grads['y'] = df(params['x'], params['y'])
        optimizer.update(params, grads)


    x = np.arange(-10, 10, 0.01)
    y = np.arange(-5, 5, 0.01)

    X, Y = np.meshgrid(x, y)
    Z = f(X, Y)

    # 외곽선 단순화
    mask = Z > 7
    Z[mask] = 0

    # 그래프 그리기
    plt.subplot(2, 2, idx)
    idx += 1
    plt.plot(x_history, y_history, 'o-', color="red")
    plt.contour(X, Y, Z)
    plt.ylim(-10, 10)
    plt.xlim(-10, 10)
    plt.plot(0, 0, '+')
    #colorbar()
    #spring()
    plt.title(key)
    plt.xlabel("x")
    plt.ylabel("y")

plt.show()
```

6.1.7 어느 갱신 방법을 이용할 것인가?

이들 네 기법의 결과를 비교

그림 6-8 최적화 기법 비교 : SGD, 모멘텀, AdaGrad, Adam



6.1.8 MNIST 데이터셋으로 본 갱신 방법 비교

```
import os
import sys
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from common.util import smooth_curve
from common.multi_layer_net import MultiLayerNet
from common.optimizer import *


# 0. MNIST 데이터 읽기==========
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)

train_size = x_train.shape[0]
batch_size = 128
max_iterations = 2000


# 1. 실험용 설정==========
optimizers = {}
optimizers['SGD'] = SGD()
optimizers['Momentum'] = Momentum()
optimizers['AdaGrad'] = AdaGrad()
optimizers['Adam'] = Adam()
#optimizers['RMSprop'] = RMSprop()

networks = {}
train_loss = {}
for key in optimizers.keys():
    networks[key] = MultiLayerNet(
        input_size=784, hidden_size_list=[100, 100, 100, 100],
        output_size=10)
    train_loss[key] = []
```

ch06/optimizer_compare_mnist.py

6.1.8 MNIST 데이터셋으로 본 갱신 방법 비교

```
# 2. 훈련 시작==========
for i in range(max_iterations):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    for key in optimizers.keys():
        grads = networks[key].gradient(x_batch, t_batch)
        optimizers[key].update(networks[key].params, grads)

        loss = networks[key].loss(x_batch, t_batch)
        train_loss[key].append(loss)

    if i % 100 == 0:
        print( "===========" + "iteration:" + str(i) + "===========")
        for key in optimizers.keys():
            loss = networks[key].loss(x_batch, t_batch)
            print(key + ":" + str(loss))


# 3. 그래프 그리기==========
markers = {"SGD": "o", "Momentum": "x", "AdaGrad": "s", "Adam": "D"}
x = np.arange(max_iterations)
for key in optimizers.keys():
    plt.plot(x, smooth_curve(train_loss[key]), marker=markers[key], markevery=100, label=key)
plt.xlabel("iterations")
plt.ylabel("loss")
plt.ylim(0, 1)
plt.legend()
plt.show()
```

6.1.8 MNIST 데이터셋으로 본 갱신 방법 비교

SGD
Momentum
AdaGrad
Adam

loss
1.0
0.8
0.6
0.4
0.2
0.0

0 250 500 750 1000 1250 1500 1750 2000
iterations

6.2 가중치의 초깃값

6.2.1 초깃값을 0으로 하면?

이제부터 오버피팅을 억제해 범용 성능을 높이는 테크닉인 가중치 감소weight decay 기법을 소개

가중치 감소는 간단히 말하자면 가중치 매개변수의 값이 작아지도록 학습하는 방법.
가중치 값을 작게 하여 오버피팅이 일어나지 않게 하는 것이다.

6.2.2 은닉층의 활성화값 분포

은닉층의 활성화값(활성화 함수의 출력 데이터)*의 분포를 관찰하면 중요한 정보를 얻을 수있다.

ch06/weight_init_activation_histogram.py

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


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


def ReLU(x):
    return np.maximum(0, x)


def tanh(x):
    return np.tanh(x)

input_data = np.random.randn(1000, 100)  # 1000개의 데이터
node_num = 100  # 각 은닉층의 노드(뉴런) 수
hidden_layer_size = 5  # 은닉층이 5개
activations = {}  # 이곳에 활성화 결과를 저장

x = input_data

for i in range(hidden_layer_size):
    if i != 0:
        x = activations[i-1]

    # 초깃값을 다양하게 바꿔가며 실험해보자!
    w = np.random.randn(node_num, node_num) * 1
    # w = np.random.randn(node_num, node_num) * 0.01
    # w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)
    # w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)

    a = np.dot(x, w)

    # 활성화 함수도 바꿔가며 실험해보자!
    z = sigmoid(a)
    # z = ReLU(a)
    # z = tanh(a)

    activations[i] = z

# 히스토그램 그리기
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    if i != 0: plt.yticks([], [])
    # plt.xlim(0.1, 1)
    # plt.ylim(0, 7000)
    plt.hist(a.flatten(), 30, range=(0,1))
plt.show()
```

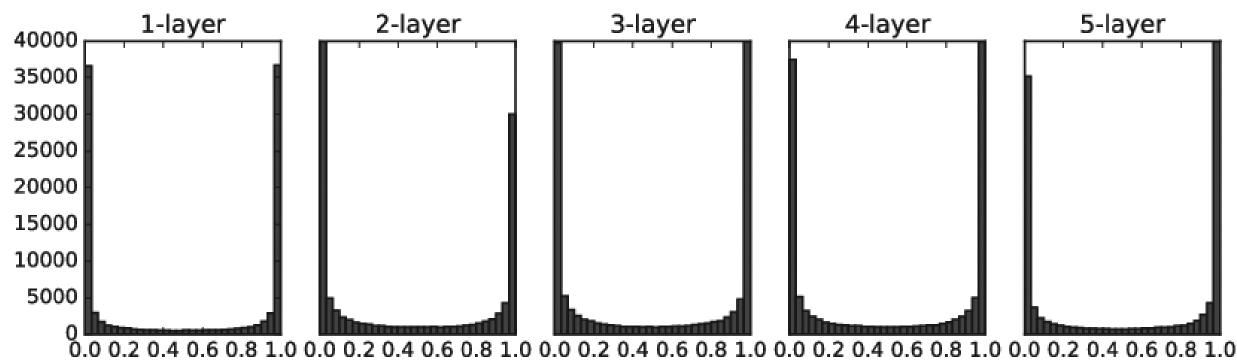


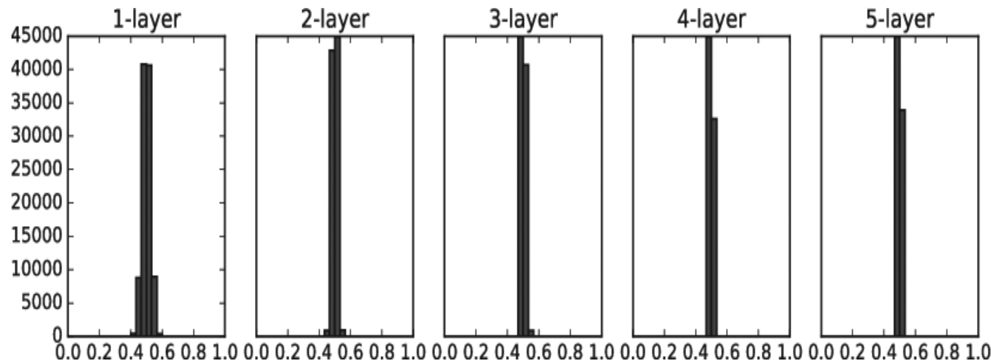
그림 **6-10** 가중치를 표준편차가 1인 정규분포로 초기화할 때의 각 층의 활성화값 분포

데이터가0과 1에 치우쳐 분포하게 되면 역전파의 기울기 값이 점점 작아지다가 사라진다.
이것이 기울기 소실gradient vanishing이라 알려진 문제이다

6.2.2 은닉층의 활성화값 분포

```
# w = np.random.randn(node_num, node_num) * 1
w = np.random.randn(node_num, node_num) * 0.01
```

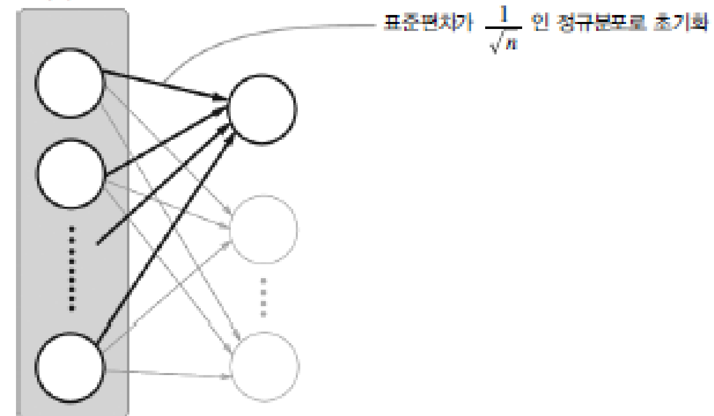
그림 6-11 가중치를 표준편차가 0.01인 정규분포로 초기화할 때의 각 층의 활성화값 분포



활성화값들이 치우치면 표현력을 제한한다는 관점에서 문제가 된다

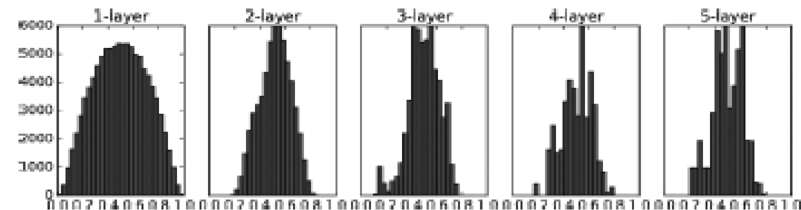
가중치 초깃값인, 일명 **Xavier** 초깃값을 써보겠다

그림 6-12 Xavier 초깃값 : 초깃값의 표준편차가 $\frac{1}{\sqrt{n}}$ 이 되도록 설정 (n은 앞 층의 노드 수)




```
node_num = 100 # 앞 층의 노드 수
w = np.random.randn(node_num, node_num) / np.sqrt(node_num)
```

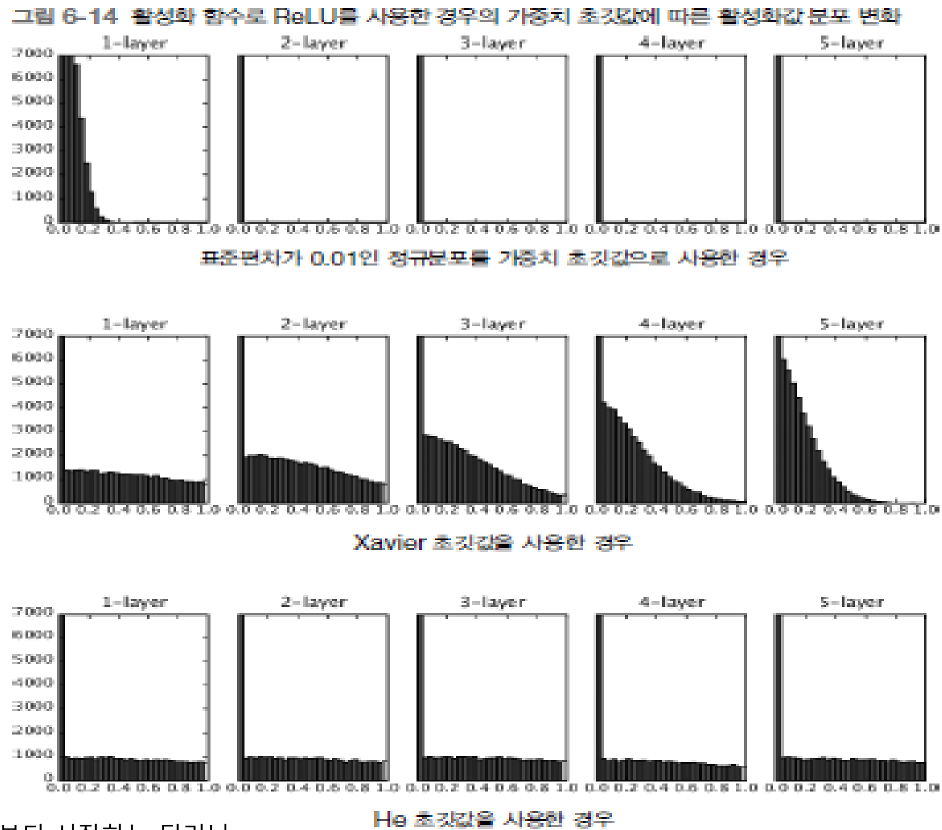
그림 6-13 가중치의 초깃값으로 'Xavier 초깃값'을 이용할 때의 각 층의 활성화값 분포



6.2.3 ReLU를 사용할 때의 가중치 초깃값

Xavier 초깃값은 활성화 함수가 선형인 것을 전제로 이끈 결과이다. Sigmoid 함수와 tanh 함수는 좌우 대칭이라 중앙 부근이 선형인 함수로 볼 수 있다

이 특화된초깃값을 찾아낸 카이밍 히Kaiming He의 이름을 따 **He** 초깃값[10]이라 한다



6.2.4 MNIST 데이터셋으로 본 가중치 초깃값 비교

ch06/weight_init_compare.py

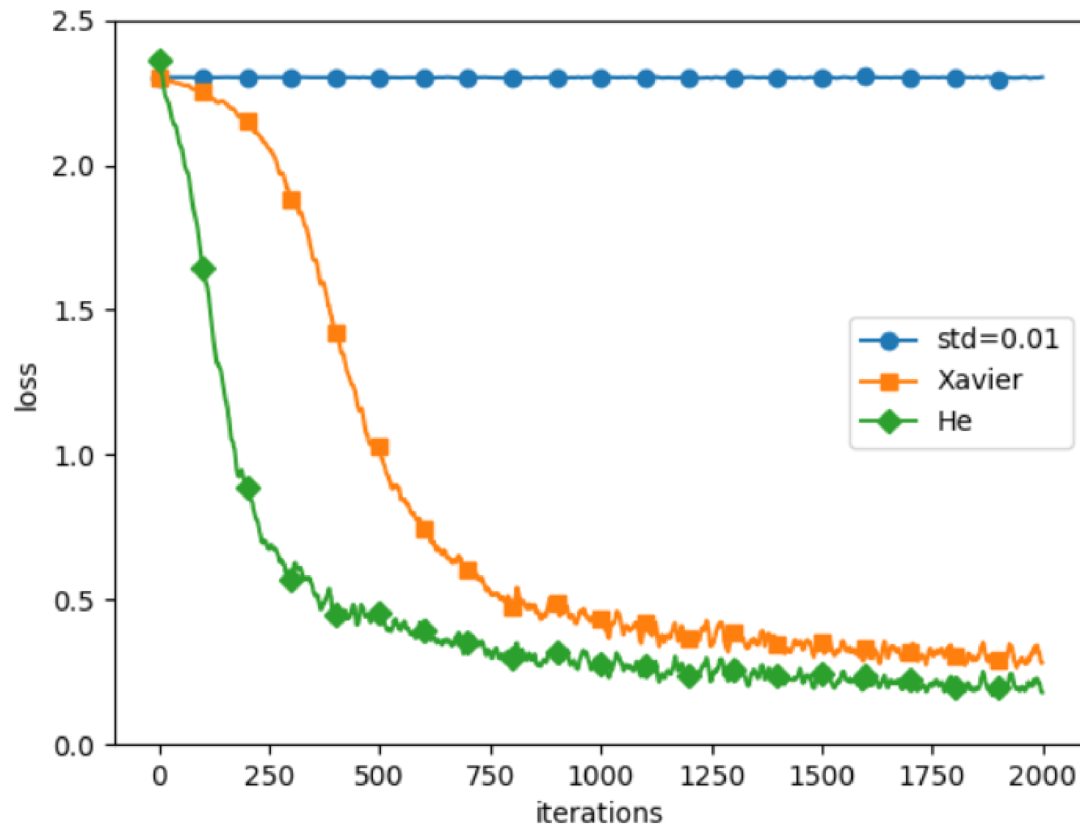
```
import os
import sys

sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from common.util import smooth_curve
from common.multi_layer_net import MultiLayerNet
from common.optimizer import SGD


# 0. MNIST 데이터 읽기==========
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)

train_size = x_train.shape[0]
batch_size = 128
max_iterations = 2000


# 1. 실험용 설정==========
weight_init_types = {'std=0.01': 0.01, 'Xavier': 'sigmoid', 'He': 'relu'}
optimizer = SGD(lr=0.01)

networks = {}
train_loss = {}
for key, weight_type in weight_init_types.items():
    networks[key] = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100],
                                  output_size=10, weight_init_std=weight_type)
    train_loss[key] = []
```

6.2.4 MNIST 데이터셋으로 본 가중치 초깃값 비교

ch06/weight_init_compare.py

```
# 2. 훈련 시작==========
for i in range(max_iterations):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    for key in weight_init_types.keys():
        grads = networks[key].gradient(x_batch, t_batch)
        optimizer.update(networks[key].params, grads)

        loss = networks[key].loss(x_batch, t_batch)
        train_loss[key].append(loss)

    if i % 100 == 0:
        print("===========" + "iteration:" + str(i) + "===========")
        for key in weight_init_types.keys():
            loss = networks[key].loss(x_batch, t_batch)
            print(key + ":" + str(loss))


# 3. 그래프 그리기==========
markers = {'std=0.01': 'o', 'Xavier': 's', 'He': 'D'}
x = np.arange(max_iterations)
for key in weight_init_types.keys():
    plt.plot(x, smooth_curve(train_loss[key]), marker=markers[key], markevery=100, label=key)
plt.xlabel("iterations")
plt.ylabel("loss")
plt.ylim(0, 2.5)
plt.legend()
plt.show()
```

6.2.4 MNIST 데이터셋으로 본 가중치 초깃값 비교

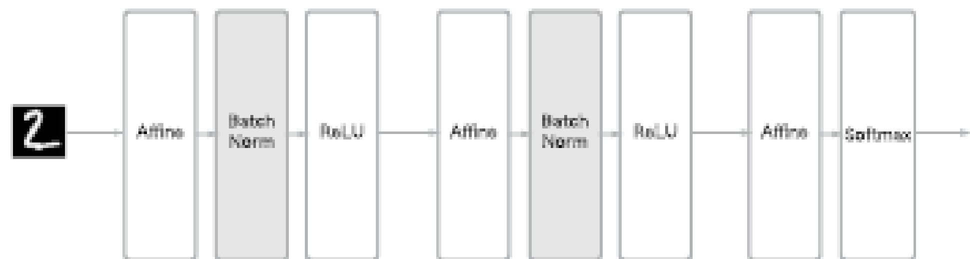


6.3 배치 정규화

6.3.1 배치 정규화 알고리즘

- 학습을 빨리 진행할 수 있다(학습 속도 개선).
- 초깃값에 크게 의존하지 않는다(골치 아픈 초깃값 선택 장애여 안녕!).
- 오버피팅을 억제한다(드롭아웃 등의 필요성 감소).

그림 6-16 배치 정규화를 사용한 신경망의 예



$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2$$

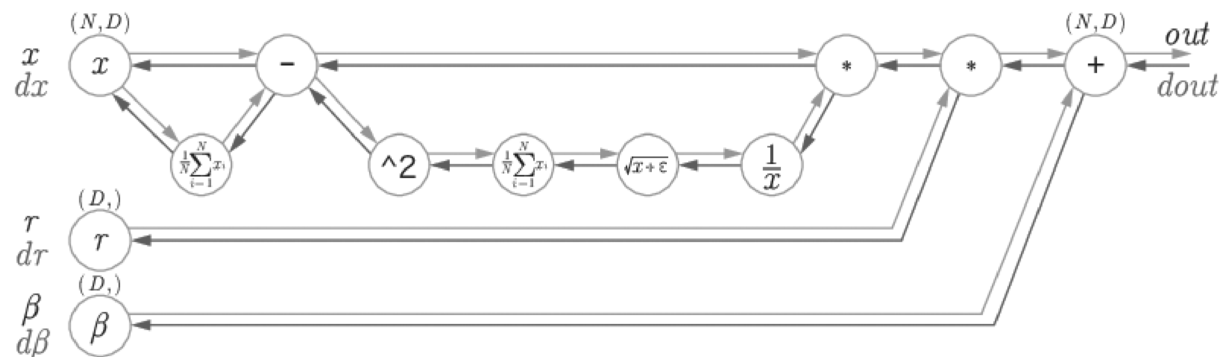
$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

[식 6.7]

$$y_i \leftarrow \gamma \hat{x}_i + \beta$$

[식 6.8]

그림 6-17 배치 정규화의 계산 그래프[13]



```python
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from common.multi_layer_net_extend import MultiLayerNetExtend
from common.optimizer import SGD, Adam

def __train(weight_init_std):
    bn_network = MultiLayerNetExtend(input_size=784, hidden_size_list=[100, 100, 100, 100, 100], output_size=10,
                                    weight_init_std=weight_init_std, use_batchnorm=True)
    network = MultiLayerNetExtend(input_size=784, hidden_size_list=[100, 100, 100, 100, 100], output_size=10,
                                weight_init_std=weight_init_std)
    optimizer = SGD(lr=learning_rate)

    train_acc_list = []
    bn_train_acc_list = []

    iter_per_epoch = max(train_size / batch_size, 1)
    epoch_cnt = 0

    for i in range(1000000000):
        batch_mask = np.random.choice(train_size, batch_size)
        x_batch = x_train[batch_mask]
        t_batch = t_train[batch_mask]

        for _network in (bn_network, network):
            grads = _network.gradient(x_batch, t_batch)
            optimizer.update(_network.params, grads)

        if i % iter_per_epoch == 0:
            train_acc = network.accuracy(x_train, t_train)
            bn_train_acc = bn_network.accuracy(x_train, t_train)
            train_acc_list.append(train_acc)
            bn_train_acc_list.append(bn_train_acc)

            print("epoch:" + str(epoch_cnt) + " | " + str(train_acc) + " - " + str(bn_train_acc))

            epoch_cnt += 1
            if epoch_cnt >= max_epochs:
                break

    return train_acc_list, bn_train_acc_list
```

```python
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)

# 학습 데이터를 줄임
x_train = x_train[:1000]
t_train = t_train[:1000]

max_epochs = 20
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.01

# 그래프 그리기==========
weight_scale_list = np.logspace(0, -4, num=16)
x = np.arange(max_epochs)

for i, w in enumerate(weight_scale_list):
    print( "============== " + str(i+1) + "/16" + " ==============")
    train_acc_list, bn_train_acc_list = __train(w)

    plt.subplot(4,4,i+1)
    plt.title("W:" + str(w))
    if i == 15:
        plt.plot(x, bn_train_acc_list, label='Batch Normalization', markevery=2)
        plt.plot(x, train_acc_list, linestyle = "--", label='Normal(without BatchNorm)', markevery=2)
    else:
        plt.plot(x, bn_train_acc_list, markevery=2)
        plt.plot(x, train_acc_list, linestyle="--", markevery=2)

    plt.ylim(0, 1.0)
    if i % 4:
        plt.yticks([])
    else:
        plt.ylabel("accuracy")
    if i < 12:
        plt.xticks([])
    else:
        plt.xlabel("epochs")
    plt.legend(loc='lower right')

plt.show()
```

6.3.2 배치 정규화의 효과

그림 6-18 배치 정규화의 효과 : 배치 정규화가 학습 속도를 높인다.

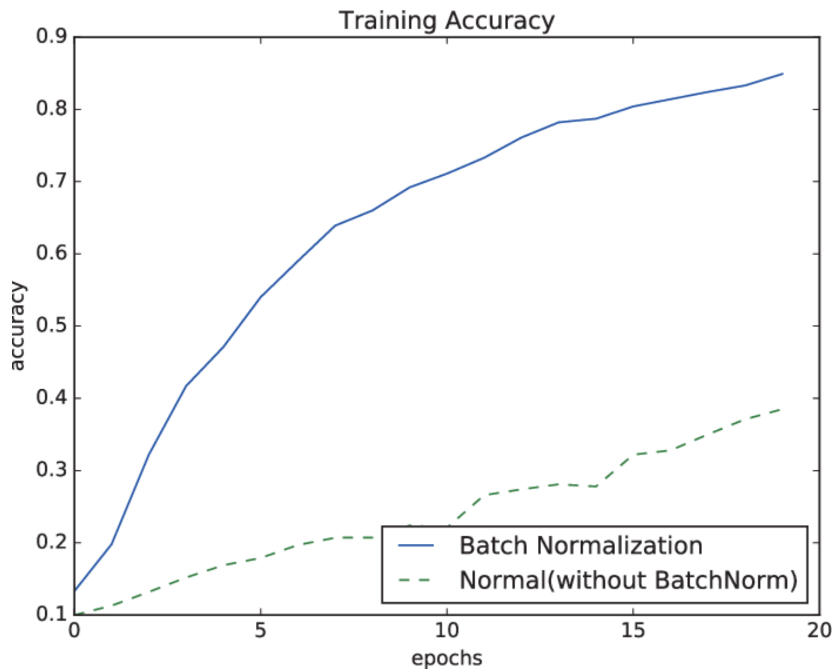
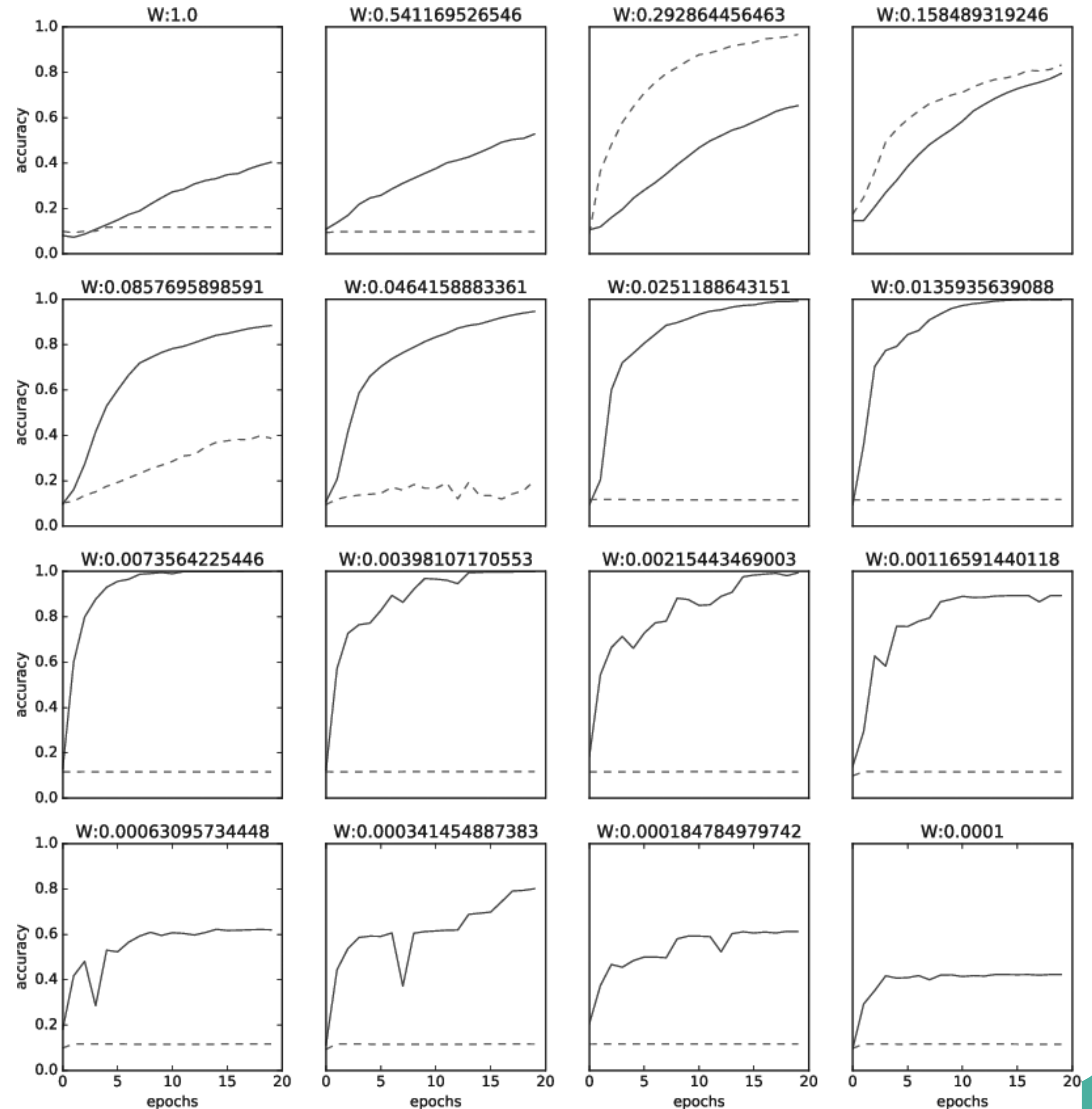


그림 6-19 실선이 배치 정규화를 사용한 경우, 점선이 사용하지 않은 경우 :



6.4.1 오버피팅

- 매개변수가 많고 표현력이 높은 모델
- 훈련 데이터가 적용

ch06/overfit_weight_decay.py

```
import os
import sys

sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from common.multi_layer_net import MultiLayerNet
from common.optimizer import SGD

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

# 오버피팅을 재현하기 위해 학습 데이터 수를 줄임
x_train = x_train[:300]
t_train = t_train[:300]

# weight decay (가중치 감쇠) 설정 =======================
#weight_decay_lambda = 0 # weight decay를 사용하지 않을 경우
weight_decay_lambda = 0.1
# ====================================================

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_size=10)
optimizer = SGD(lr=0.01) # 학습률이 0.01인 SGD로 매개변수 갱신

max_epochs = 201
train_size = x_train.shape[0]
batch_size = 100

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

iter_per_epoch = max(train_size / batch_size, 1)
epoch_cnt = 0
```

6.4 바른 학습을 위해

6.4.1 오버피팅

```
for i in range(1000000000):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    grads = network.gradient(x_batch, t_batch)
    optimizer.update(network.params, grads)

    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("epoch:" + str(epoch_cnt) + ", train acc:" + str(train_acc) + ", test acc:" + str(test_acc))

        epoch_cnt += 1
        if epoch_cnt >= max_epochs:
            break


# 그래프 그리기==========
markers = {'train': 'o', 'test': 's'}
x = np.arange(max_epochs)
plt.plot(x, train_acc_list, marker='o', label='train', markevery=10)
plt.plot(x, test_acc_list, marker='s', label='test', markevery=10)
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```

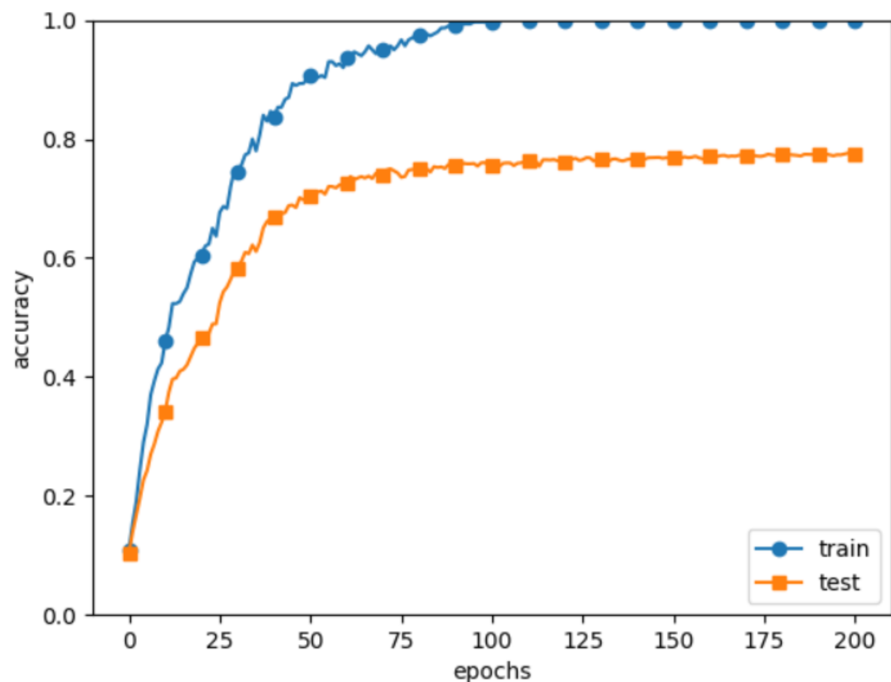


그림 **6-20** 훈련 데이터(train)와 시험 데이터(test)의 에폭별 정확도 추이

6.4.2 가중치 감소

오버피팅 억제용으로 예로부터 많이 이용해온 방법 중 가중치 감소weight decay라는 것이 있다.

common/multi_layer_net.py

```python
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
from collections import OrderedDict
from common.layers import *
from common.gradient import numerical_gradient


class MultiLayerNet:
    """완전연결 다층 신경망

    Parameters
    ----------
    input_size : 입력 크기 (MNIST의 경우엔 784)
    hidden_size_list : 각 은닉층의 뉴런 수를 담은 리스트 (e.g. [100, 100, 100])
    output_size : 출력 크기 (MNIST의 경우엔 10)
    activation : 활성화 함수 - 'relu' 혹은 'sigmoid'
    weight_init_std : 가중치의 표준편차 지정 (e.g. 0.01)
        'relu'나 'he'로 지정하면 'He 초깃값'으로 설정
        'sigmoid'나 'xavier'로 지정하면 'Xavier 초깃값'으로 설정
    weight_decay_lambda : 가중치 감소(L2 법칙)의 세기
    """
```

```python
def loss(self, x, t):
    """손실 함수를 구한다.

    Parameters
    ----------
    x : 입력 데이터
    t : 정답 레이블

    Returns
    -------
    손실 함수의 값
    """
    y = self.predict(x)

    weight_decay = 0
    for idx in range(1, self.hidden_layer_num + 2):
        W = self.params['W' + str(idx)]
        weight_decay += 0.5 * self.weight_decay_lambda * np.sum(W ** 2)

    return self.last_layer.forward(y, t) + weight_decay
```

6.4.2 가중치 감소

common/multi_layer_net.py

```
def gradient(self, x, t):
    """기울기를 구한다(오차역전파법).

    Parameters
    ----------
    x : 입력 데이터
    t : 정답 레이블

    Returns
    -------
    각 층의 기울기를 담은 딕셔너리(dictionary) 변수
        grads['W1']、grads['W2']、... 각 층의 가중치
        grads['b1']、grads['b2']、... 각 층의 편향
    """
    # forward
    self.loss(x, t)

    # backward
    dout = 1
    dout = self.last_layer.backward(dout)

    layers = list(self.layers.values())
    layers.reverse()
    for layer in layers:
        dout = layer.backward(dout)

    # 결과 저장
    grads = {}
    for idx in range(1, self.hidden_layer_num+2):
        grads['W' + str(idx)] = self.layers['Affine' + str(idx)].dW + self.weight_decay_lambda * self.layers['Affine' + str(idx)].W
        grads['b' + str(idx)] = self.layers['Affine' + str(idx)].db

    return grads
```

6.4 바른 학습을 위해

6.4.2 가중치 감소

ch06/overfit_weight_decay.py

```
import os
import sys

sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from common.multi_layer_net import MultiLayerNet
from common.optimizer import SGD

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

# 오버피팅을 재현하기 위해 학습 데이터 수를 줄임
x_train = x_train[:300]
t_train = t_train[:300]

# weight decay (가중치 감쇠) 설정 =======================
#weight_decay_lambda = 0 # weight decay를 사용하지 않을 경우
weight_decay_lambda = 0.1
# ====================================================

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_size=10,
                        weight_decay_lambda=weight_decay_lambda)
optimizer = SGD(lr=0.01) # 학습률이 0.01인 SGD로 매개변수 갱신

max_epochs = 201
train_size = x_train.shape[0]
batch_size = 100

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

iter_per_epoch = max(train_size / batch_size, 1)
epoch_cnt = 0
```

6.4.2 가중치 감소

```
for i in range(1000000000):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    grads = network.gradient(x_batch, t_batch)
    optimizer.update(network.params, grads)

    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("epoch:" + str(epoch_cnt) + ", train acc:" + str(train_acc) + ", test acc:" + str(test_acc))

        epoch_cnt += 1
        if epoch_cnt >= max_epochs:
            break


# 그래프 그리기==========
markers = {'train': 'o', 'test': 's'}
x = np.arange(max_epochs)
plt.plot(x, train_acc_list, marker='o', label='train', markevery=10)
plt.plot(x, test_acc_list, marker='s', label='test', markevery=10)
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```

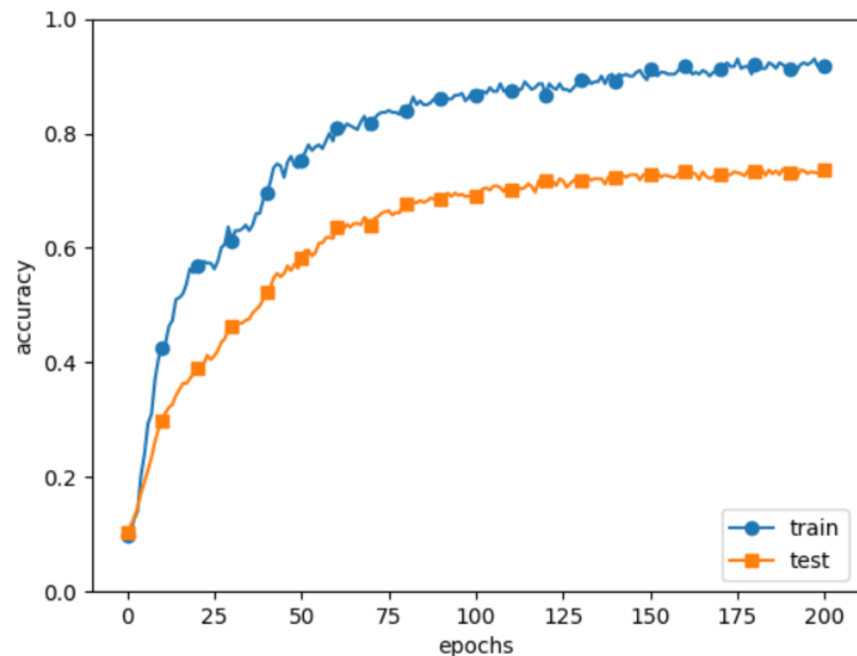
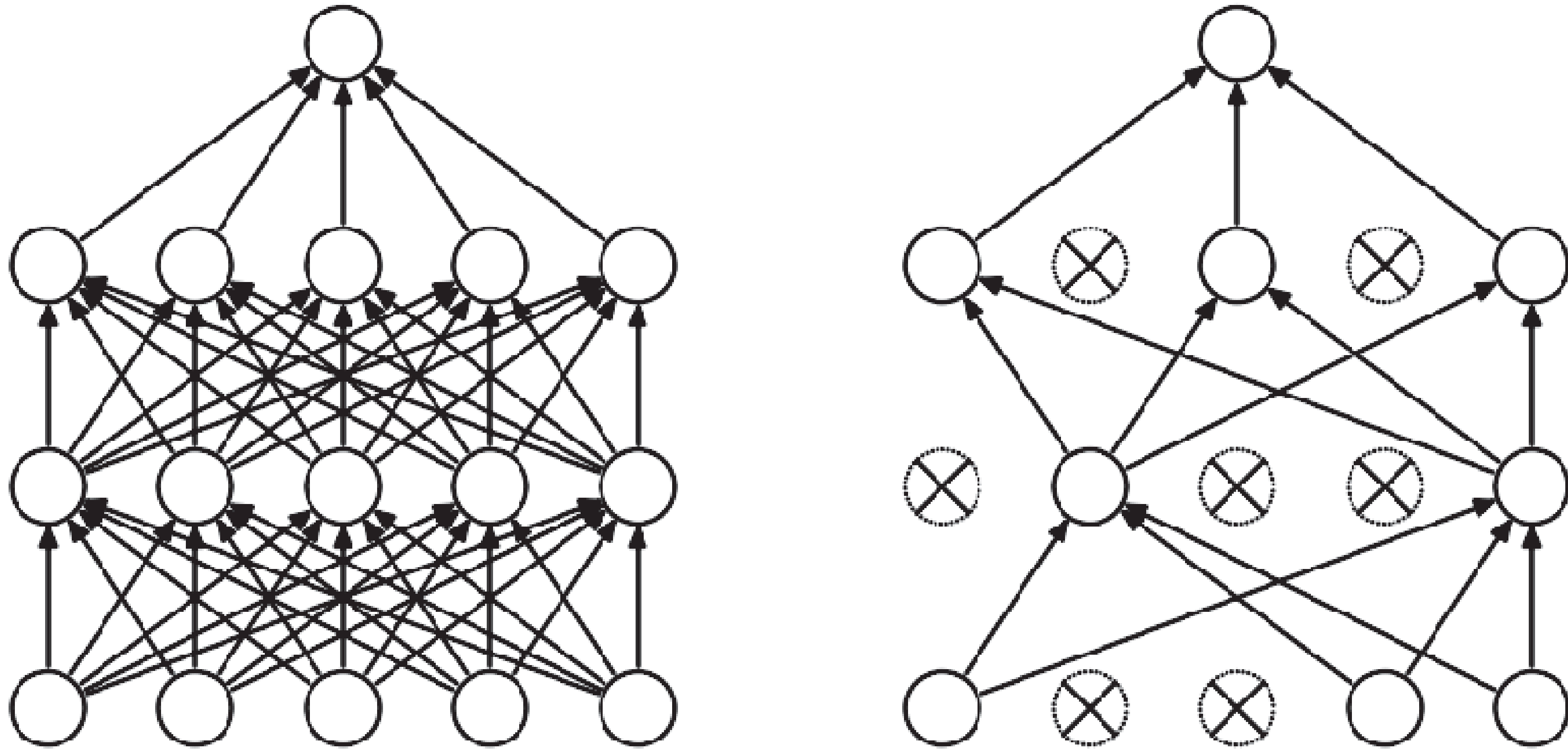


그림 **6-21 가중치 감소를 이용한** 훈련 데이터(train)와 시험 데이터(test)의 에폭별 정확도 추이

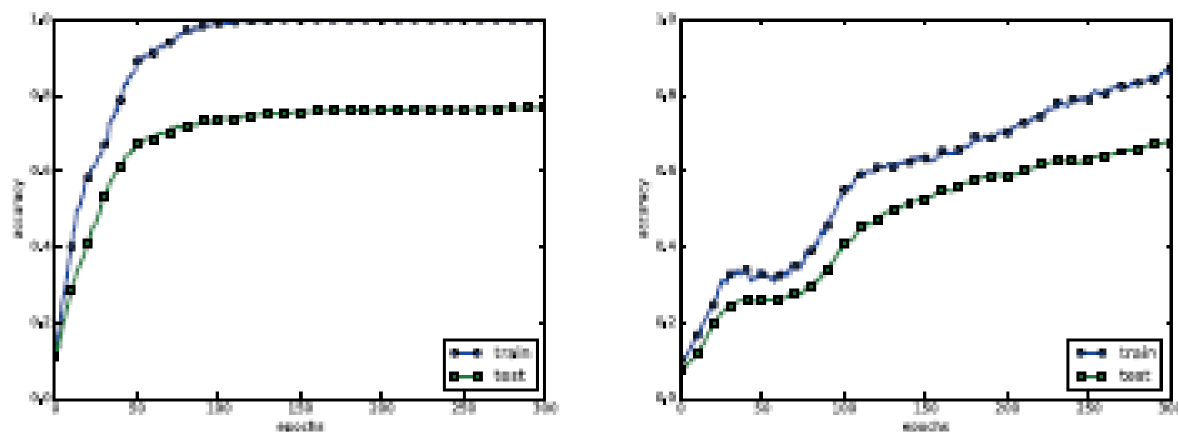
6.4.3 드롭아웃

그림 **6-22** 드롭아웃의 개념(문헌[14]에서 인용) : 왼쪽이 일반적인 신경망, 오른쪽이 드롭아웃을 적용한 신경망. 드롭아웃은 뉴런을 무작위로 선택해 삭제하여 신호 전달을 차단한다.



6.4.3 드롭아웃

그림 6-23 왼쪽은 드롭아웃 없이, 오른쪽은 드롭아웃을 적용한 결과 (dropout_ratio = 0.15)



common/layers.py

```python
class Dropout:
    """
    http://arxiv.org/abs/1207.0580
    """
    def __init__(self, dropout_ratio=0.5):
        self.dropout_ratio = dropout_ratio
        self.mask = None

    def forward(self, x, train_flg=True):
        if train_flg:
            self.mask = np.random.rand(*x.shape) > self.dropout_ratio
            return x * self.mask
        else:
            return x * (1.0 - self.dropout_ratio)

    def backward(self, dout):
        return dout * self.mask
```

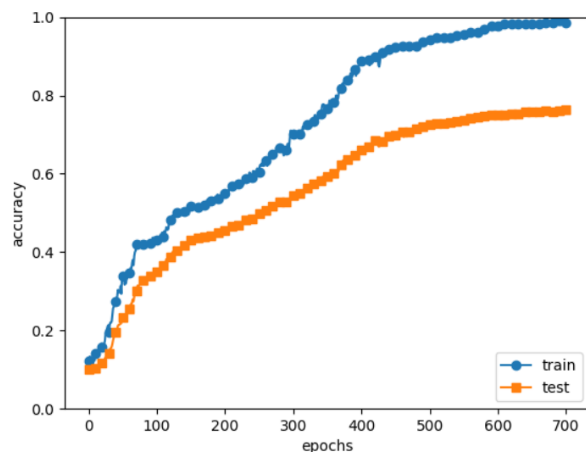
x의 크기가 결정되지 않아서...

self.mask에 삭제할 뉴런을 false로 표시

Test 시 적용

순전파 시 통과 뉴런은 역전파 시 신호 통과, 그렇지 않으면 신호 차단

```
import os
import sys
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from common.multi_layer_net_extend import MultiLayerNetExtend
from common.trainer import Trainer

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

# 오버피팅을 재현하기 위해 학습 데이터 수를 줄임
x_train = x_train[:300]
t_train = t_train[:300]

# 드롭아웃 사용 유무와 비율 설정 ========================
use_dropout = True  # 드롭아웃을 쓰지 않을 때는 False
dropout_ratio = 0.2
# ====================================================

network = MultiLayerNetExtend(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100],
                              output_size=10, use_dropout=use_dropout, dropout_ration=dropout_ratio)
trainer = Trainer(network, x_train, t_train, x_test, t_test,
                  epochs=301, mini_batch_size=100,
                  optimizer='sgd', optimizer_param={'lr': 0.01}, verbose=True)
trainer.train()

train_acc_list, test_acc_list = trainer.train_acc_list, trainer.test_acc_list

# 그래프 그리기==========
markers = {'train': 'o', 'test': 's'}
x = np.arange(len(train_acc_list))
plt.plot(x, train_acc_list, marker='o', label='train', markevery=10)
plt.plot(x, test_acc_list, marker='s', label='test', markevery=10)
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```



6.5 적절한 하이퍼파라미터 값 찾기

하이퍼파라미터: 각 층의 뉴런 수, 배치 크기, 학습률과 가중치 감소 등

6.5.1 검증 데이터

하이퍼파라미터를 조정할 때는 하이퍼파라미터 전용 확인 데이터가 필요하다.
하이퍼파라미터 조정용 데이터를 일반적으로 검증 데이터 validation data 라고 부른다

훈련 데이터(training set): 매개변수 학습
검증 데이터(validation set): 하이퍼 파라미터 성능 평가
시험 데이터(test set): 신경망의 범용 성능 평가

```
(x_train, t_train), (x_test, t_test) = load_mnist()

# 훈련 데이터를 뒤섞는다.
x_train, t_train = shuffle_dataset(x_train, t_train)

# 20%를 검증 데이터로 분할
validation_rate = 0.20
validation_num = int(x_train.shape[0] * validation_rate)

x_val = x_train[:validation_num]
t_val = t_train[:validation_num]
x_train = x_train[validation_num:]
t_train = t_train[validation_num:]
```

common/util.py

```
def shuffle_dataset(x, t):
    """데이터셋을 뒤섞는다.

    Parameters
    ----------
    x : 훈련 데이터
    t : 정답 레이블

    Returns
    -------
    x, t : 뒤섞은 훈련 데이터와 정답 레이블
    """
    permutation = np.random.permutation(x.shape[0])
    x = x[permutation,:] if x.ndim == 2 else x[permutation,:,:,:]
    t = t[permutation]

    return x, t
```

6.5.2 하이퍼파라미터 최적화

- 0단계
하이퍼파라미터 값의 범위를 설정.

- 1단계
설정된 범위에서 하이퍼파라미터의 값을 무작위로 추출.

- 2단계
1단계에서 샘플링한 하이퍼파라미터 값을 사용하여 학습하고, 검증 데이터로 정확도를 평가(단, 에폭은 작게 설정).

- 3단계
1단계와 2단계를 특정 횟수(100회 등) 반복하며, 그 정확도의 결과를 보고 하이퍼파라미터의 범위를 좁힌다.

6.5.3 하이퍼파라미터 최적화 구현하기

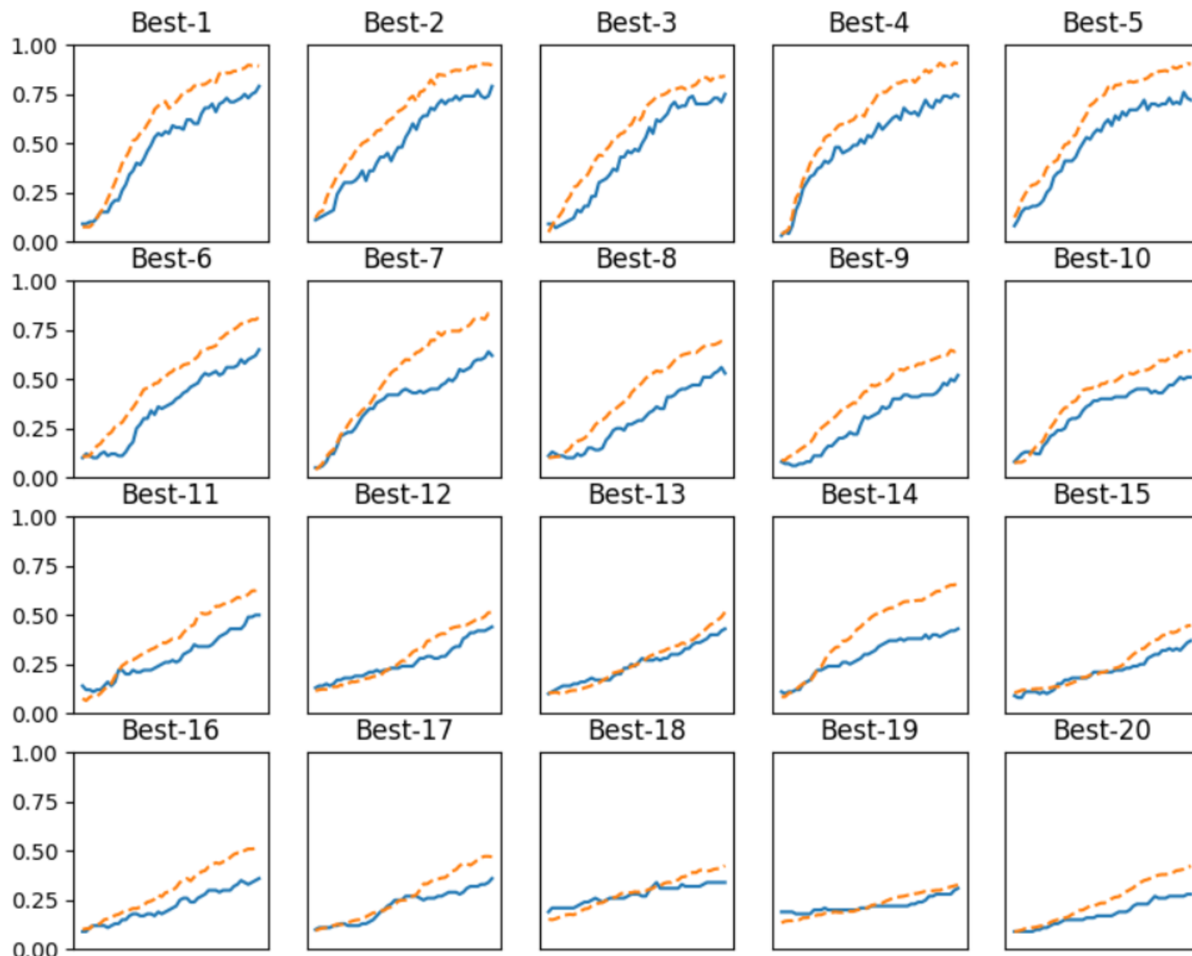
ch06/hyperparameter_optimization.py

```python
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from common.multi_layer_net import MultiLayerNet
from common.util import shuffle_dataset
from common.trainer import Trainer

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

# 결과를 빠르게 얻기 위해 훈련 데이터를 줄임
x_train = x_train[:500]
t_train = t_train[:500]

# 20%를 검증 데이터로 분할
validation_rate = 0.20
validation_num = int(x_train.shape[0] * validation_rate)
x_train, t_train = shuffle_dataset(x_train, t_train)
x_val = x_train[:validation_num]
t_val = t_train[:validation_num]
x_train = x_train[validation_num:]
t_train = t_train[validation_num:]


def __train(lr, weight_decay, epocs=50):
    network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100],
                            output_size=10, weight_decay_lambda=weight_decay)
    trainer = Trainer(network, x_train, t_train, x_val, t_val,
                      epochs=epocs, mini_batch_size=100,
                      optimizer='sgd', optimizer_param={'lr': lr}, verbose=False)
    trainer.train()

    return trainer.test_acc_list, trainer.train_acc_list
```

```python
# 하이퍼파라미터 무작위 탐색======================================
optimization_trial = 100
results_val = {}
results_train = {}
for _ in range(optimization_trial):
    # 탐색한 하이퍼파라미터의 범위 지정===============
    weight_decay = 10 ** np.random.uniform(-8, -4)
    lr = 10 ** np.random.uniform(-6, -2)
    # ================================================

    val_acc_list, train_acc_list = __train(lr, weight_decay)
    print("val acc:" + str(val_acc_list[-1]) + " | lr:" + str(lr) + ", weight decay:" + str(weight_decay))
    key = "lr:" + str(lr) + ", weight decay:" + str(weight_decay)
    results_val[key] = val_acc_list
    results_train[key] = train_acc_list

# 그래프 그리기========================================================
print("=========== Hyper-Parameter Optimization Result ===========")
graph_draw_num = 20
col_num = 5
row_num = int(np.ceil(graph_draw_num / col_num))
i = 0

for key, val_acc_list in sorted(results_val.items(), key=lambda x:x[1][-1], reverse=True):
    print("Best-" + str(i+1) + "(val acc:" + str(val_acc_list[-1]) + ") | " + key)

    plt.subplot(row_num, col_num, i+1)
    plt.title("Best-" + str(i+1))
    plt.ylim(0.0, 1.0)
    if i % 5: plt.yticks([])
    plt.xticks([])
    x = np.arange(len(val_acc_list))
    plt.plot(x, val_acc_list)
    plt.plot(x, results_train[key], "--")
    i += 1

    if i >= graph_draw_num:
        break

plt.show()
```

SECTION 06 학습 관련 기술들

6.5.3 하이퍼파라미터 최적화 구현하기




```
=========== Hyper-Parameter Optimization Result ===========
Best-1(val acc:0.79) | lr:0.007775929506396434, weight decay:5.90045828295005e-07
Best-2(val acc:0.79) | lr:0.007254343305129658, weight decay:1.1524225087770819e-07
Best-3(val acc:0.75) | lr:0.0047233458759333045, weight decay:8.526982342958834e-07
Best-4(val acc:0.74) | lr:0.009487279828549822, weight decay:2.4101815416504603e-08
Best-5(val acc:0.72) | lr:0.008334948326110887, weight decay:2.4665680405615925e-07
Best-6(val acc:0.65) | lr:0.004766760772381838, weight decay:9.477864357872687e-05
Best-7(val acc:0.62) | lr:0.005723943990980994, weight decay:1.4886747340983507e-05
Best-8(val acc:0.53) | lr:0.0040461229188121575, weight decay:1.9709929854417675e-08
Best-9(val acc:0.52) | lr:0.003756794149683187, weight decay:1.1575823320127595e-06
Best-10(val acc:0.51) | lr:0.002916555901114869, weight decay:9.983520058371334e-07
Best-11(val acc:0.5) | lr:0.0046394900261658885, weight decay:1.2905732940586724e-07
Best-12(val acc:0.44) | lr:0.0019425702812241477, weight decay:6.460292887055926e-06
Best-13(val acc:0.43) | lr:0.002577659210075172, weight decay:2.5288553496405504e-05
Best-14(val acc:0.43) | lr:0.00414809786323954, weight decay:3.2122720220744856e-06
Best-15(val acc:0.37) | lr:0.0029268534436303214, weight decay:9.672252853405284e-06
Best-16(val acc:0.36) | lr:0.002516096234743461, weight decay:2.1247725015282106e-05
Best-17(val acc:0.36) | lr:0.002283440475666735, weight decay:2.1436435980283782e-07
Best-18(val acc:0.34) | lr:0.0024578556353409054, weight decay:2.0305153910454517e-08
Best-19(val acc:0.31) | lr:0.000869751678862249, weight decay:9.227599137136632e-07
Best-20(val acc:0.28) | lr:0.001953066528491176, weight decay:3.6610855405633556e-07
```