Machine Learning

Optimizer Tuning

❖ 매개 변수 갱신

- 모멘텀 (Momentum)
  - ‘운동량’ 을 뜻하는 단어로 물리와 관계가 있음

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$$

  - W : 갱신 할 가중치
  - $\frac{\partial L}{\partial W}$: 손실함수 기울기
  - η : 학습율
  - v : 물리에서의 속도

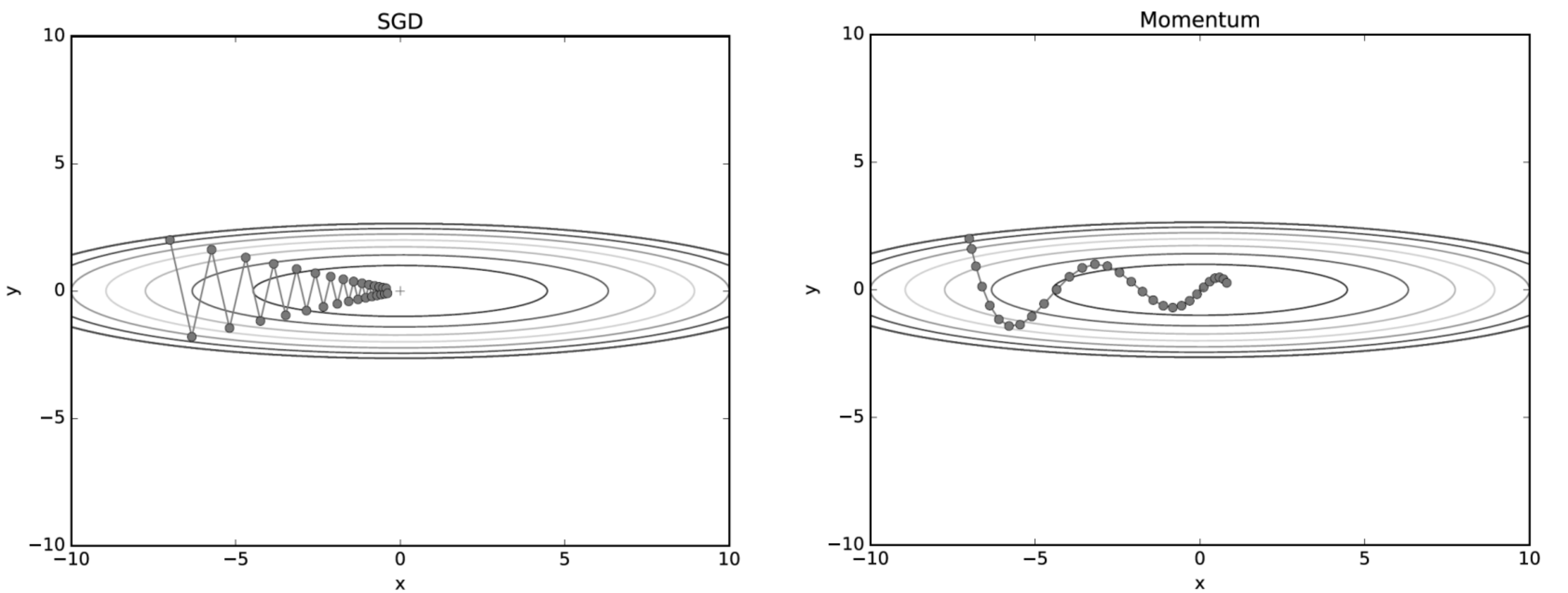
❖ 매개 변수 갱신

- **모멘텀 (Momentum)**

```
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]
```

❖ 매개 변수 갱신

- 모멘텀 (Momentum)



❖ 매개 변수 갱신

- AdaGrad
  - AdaGrad는 '각각의' 매개변수에 맞게 '맞춤형'으로 매개 변수를 갱신.

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

  - W : 갱신 할 가중치
  - $\frac{\partial L}{\partial W}$: 손실함수 기울기
  - η : 학습율
  - v : 물리에서의 속도

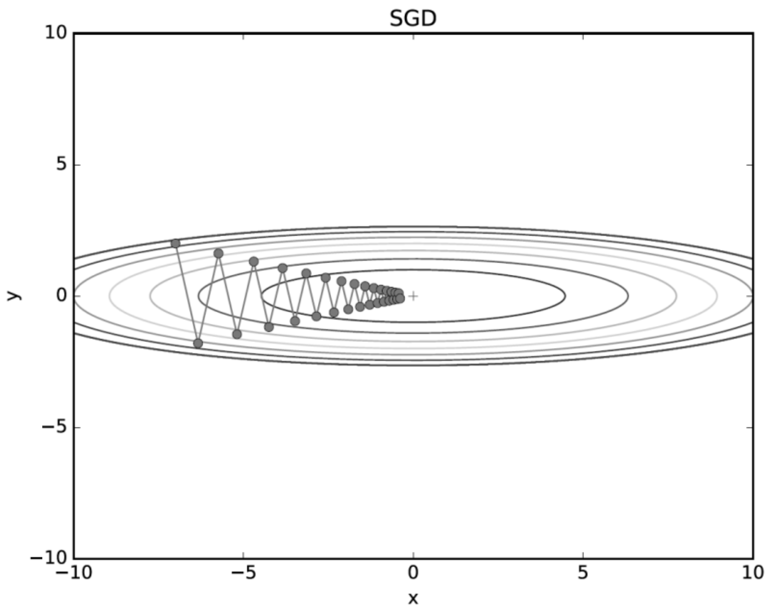
❖ 매개 변수 갱신

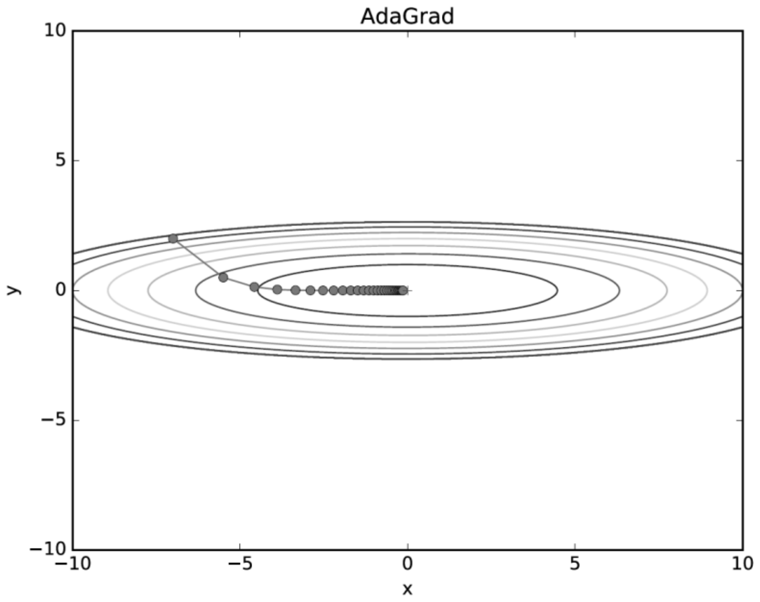
- *AdaGrad*

```
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)
```

❖ 매개 변수 갱신

- 모멘텀 (Momentum)





❖ 매개 변수 갱신

- Adam
  - **직관적으로는 모멘텀과 AdaGrad를 융합한 방법**

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta J(\theta))^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \Rightarrow \theta = \theta - \frac{\eta}{\sqrt{\hat{v}_t+\epsilon}}\hat{m}_t$$

  - m_t: **momentum처럼 지금까지 계산해온 기울기의 지수평균을 저장**
  - v_t : **AdaGrad처럼 기울기 제곱 값의 지수평균을 저장**

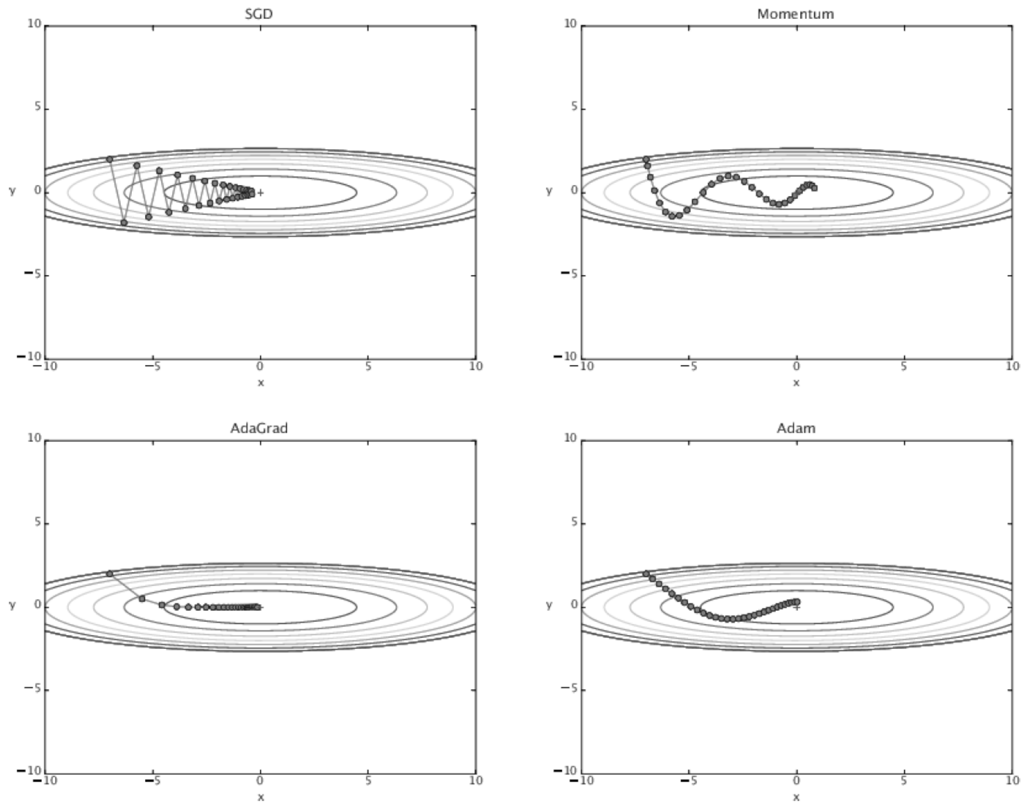
  - 보통 β_1로는 0.9, β_2로는 0.999, ϵ으로는 10^{-8} 정도의 값을 사용

❖ 매개 변수 갱신

```
class Adam:
    def __init__(self, lr=0.001, beta1=0.9, beta2=0.999):
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.iter = 0
        self.m = None
        self.v = None

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

        self.iter += 1
        lr_t  = self.lr * np.sqrt(1.0 - self.beta2**self.iter) / (1.0 - self.beta1**self.iter)

        for key in params.keys():
            self.m[key] = self.beta1*self.m[key] + (1-self.beta1)*grads[key]
            self.v[key] = self.beta2*self.v[key] + (1-self.beta2)*(grads[key]**2)
            params[key] -= lr_t * self.m[key] / (np.sqrt(self.v[key]) + 1e-7)
```

❖ 매개 변수 갱신

- Adam
  - 직관적으로는 모멘텀과 AdaGrad를 융합한 방법



❖ 매개 변수 갱신

- optimizer_compare_mnist

```
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
```

❖ 매개 변수 갱신

- optimizer_compare_mnist

```
# 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] = []
```

❖ 매개 변수 갱신

- optimizer_compare_mnist

```
# 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))
```

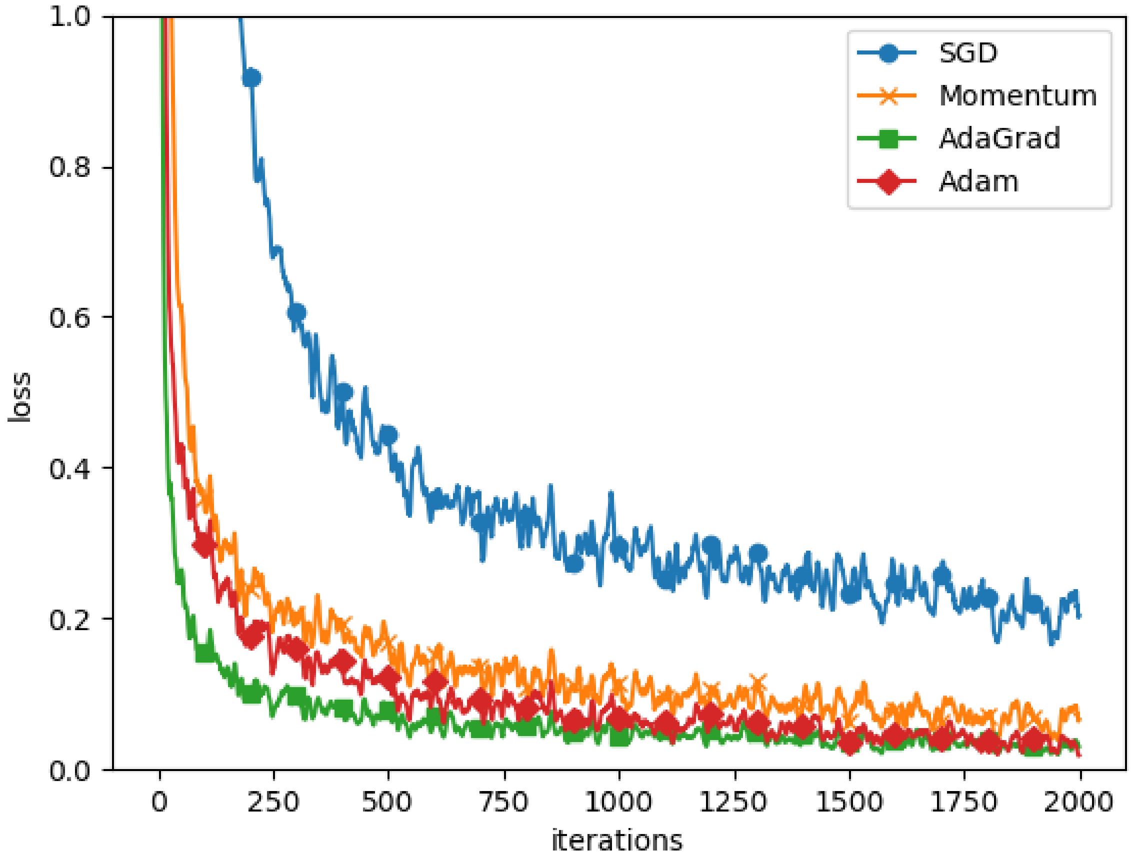
❖ 매개 변수 갱신

- optimizer_compare_mnist

```
# 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()
```

❖ 매개 변수 갱신

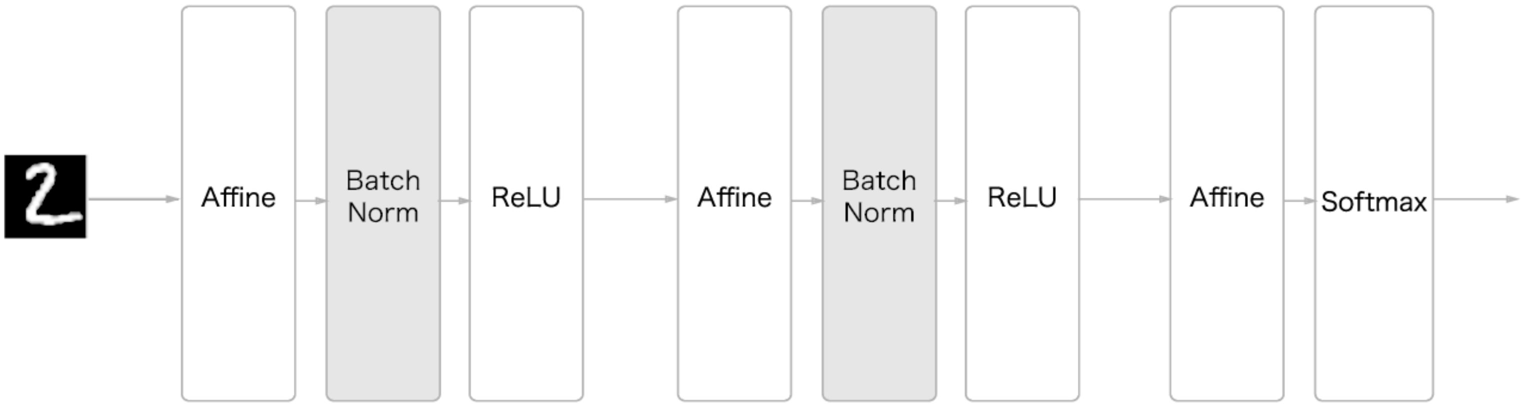
- Adam



1.0
0.8
0.6
0.4
0.2
0.0
loss
0
250
500
750
1000
1250
1500
1750
2000
iterations
SGD
Momentum
AdaGrad
Adam

❖ 배치 정규화

- 각 층에서의 활성화 값이 적당히 분포되도록 조정
- 배치 정규화의 장점
  - 학습을 빨리 진행할 수 있다(학습속도 개선)
  - 초기값에 크게 의존하지 않는다
  - 오버피팅을 억제함(드롭아웃 등의 필요성 감소)
- '배치 정규화 계층'을 신경망에 삽입



❖ 배치 정규화

- 학습 시 미니 배치를 단위로 정규화
  - 데이터 분포가 평균이 0, 분산이 1이 되도록 정규화

미니배치 평균

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

미니배치 분산

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

평균0, 분산1로 정규화

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

- $\boldsymbol{B} = \{x_1, x_2, \dots, x_m\}$: 미니 배치 집합
- $\boldsymbol{\mu}_B$: 평균
- $\boldsymbol{\sigma}_B^2$: 분산
- ε : 작은 값 (ex. 10^{-7})

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

- 정규화된 데이터에 고유한 확대scale(γ)와 이동shift(β) 변환을 수행
  - ✓ $\gamma = 1$, $\beta = 0$ 부터 시작, 학습하면서 조정

❖ 배치 정규화

- **batch_norm_test**

```
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

(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
```

❖ 배치 정규화

- **batch_norm_test**

```
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
```

❖ 배치 정규화

▪ **batch_norm_test**

```
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
```

❖ 배치 정규화

- **batch_norm_test**

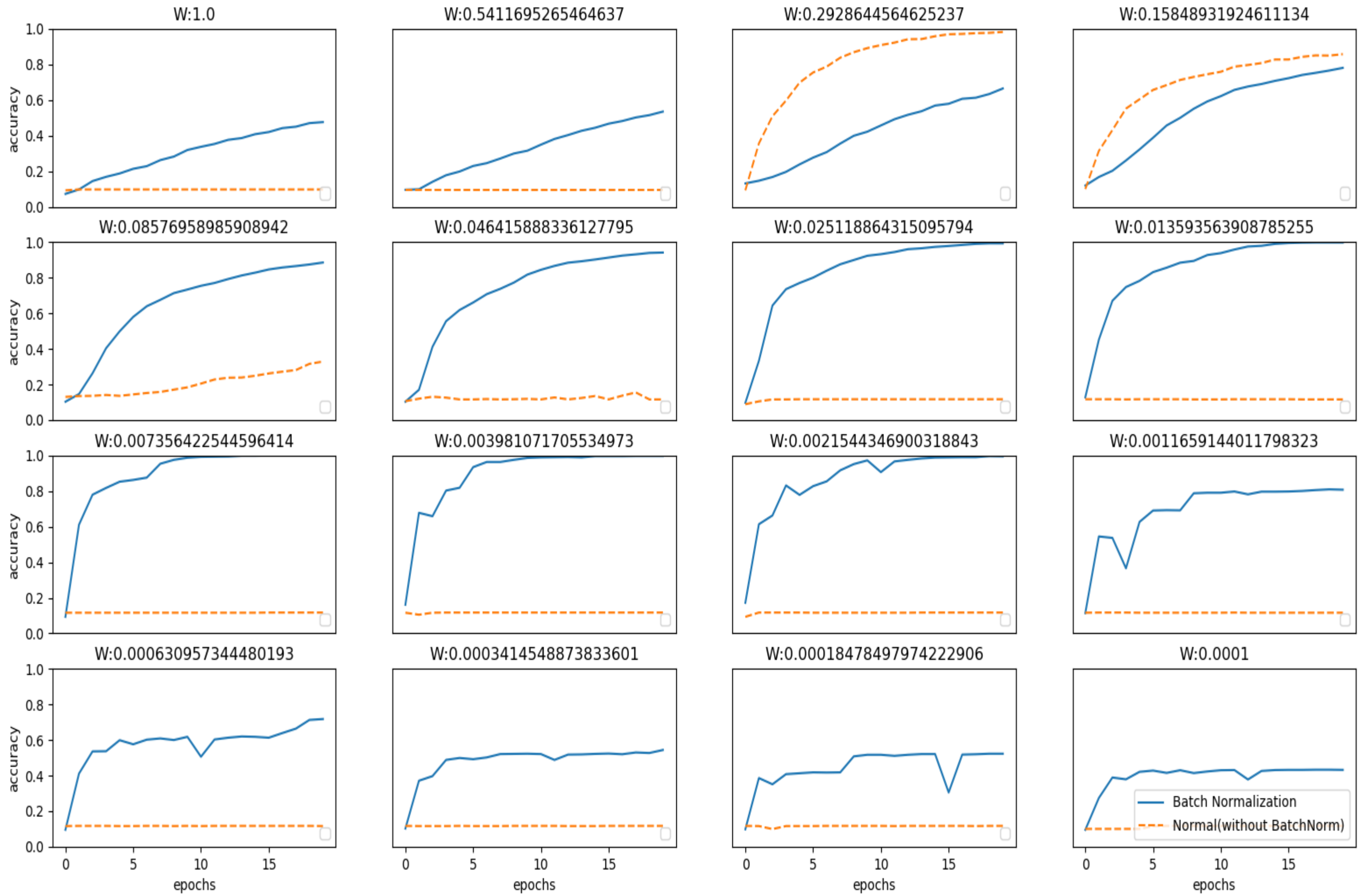
```
# 그래프 그리기==========
weight_scale_list = np.logspace(0, -4, num=16)
x = np.arange(max_epochs)

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)
```

❖ 배치 정규화

- **batch_norm_test**

```
    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()
```

❖ 배치 정규화



W:1.0
W:0.5411695265464637
W:0.2928644564625237
W:0.15848931924611134
W:0.08576958985908942
W:0.046415888336127795
W:0.025118864315095794
W:0.013593563908785255
W:0.007356422544596414
W:0.003981071705534973
W:0.0021544346900318843
W:0.0011659144011798323
W:0.0006309573444801930
W:0.00034145488738336010
W:0.00018478497974222906
W:0.0001
accuracy
epochs
Batch Normalization
Normal(without BatchNorm)

❖ 오버피팅 억제를 위한 방법

- 모델 크기 줄이기
  - 층, 뉴런의 개수 등 학습해야 할 파라미터의 개수를 줄임
- Early stopping
  - 학습을 일찍 중단
- 가중치 감소
  - 학습 파라미터의 값이 크면 그에 상응하는 큰 패널티를 부과
    - ✓L2 Regularization, L1 Regularization, L_∞ Regularization
- Dropout
  - 일부 뉴런을 꺼서 학습, 일종의 앙상블ensemble 효과를 냄
  - 학습 시 삭제할 뉴런을 무작위로 선택, 테스트 시 모든 뉴런을 사용

❖ 오버피팅 억제를 위한 방법

- **overfit_weight_decay**

```
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
# ====================================================
```

❖ 오버피팅 억제를 위한 방법

- **overfit_weight_decay**

```
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
```

❖ 오버피팅 억제를 위한 방법

- **overfit_weight_decay**

```
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
```

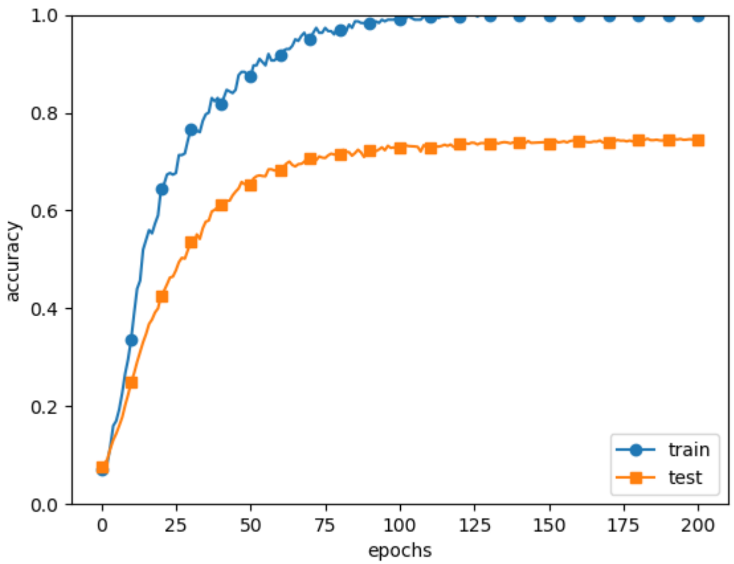
❖ 오버피팅 억제를 위한 방법

- overfit_weight_decay

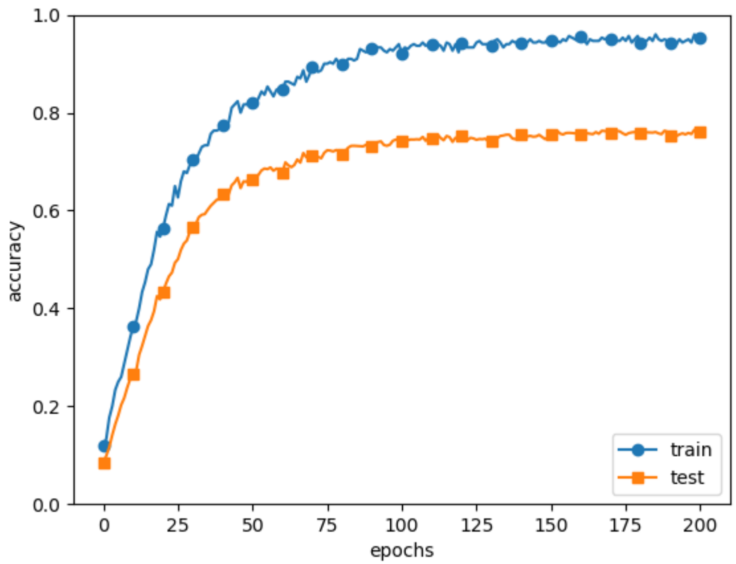
```
# 그래프 그리기==========
markers = {'train': 'o', 'test': 's'}
x = np.arange(max_epochs)
plt.plot(x, train_acc_list, marker='o', label='train', markevery=10)
plt.plot(x, test_acc_list, marker='s', label='test', markevery=10)
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower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

❖ 오버피팅 억제를 위한 방법

Weight decay 미 적용



Weight decay 적용



❖ 오버피팅 억제를 위한 방법

- **overfit_dropout**

```
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 비울 설정 ========================
use_dropout = True  # 드롭아웃을 쓰지 않을 때는 False
dropout_ratio = 0.2
# ====================================================
```

❖ 오버피팅 억제를 위한 방법

- **overfit_dropout**

```
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()
```

❖ 오버피팅 억제를 위한 방법