

# CHAPTER 6 학습 관련 기술들

신경망(딥러닝) 학습의 효율과 정확도를 높이는 방법들에 대해 학습  
다른 매개변수 업그레이드 방법  
매개변수 초기값 잘 주는 방법  
배치 정규화  
신경망 모델 오버피팅 방지하는 방법  
좋은 하이퍼 파라미터 설정 방법

# 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 초기값
  - 6.2.1 초기값을 0으로 하면?
  - 6.2.2 은닉층의 활성화값 분포
  - 6.2.3 ReLU를 사용할 때의 가중치 초기값
- 6.2.4 MNIST 데이터셋으로 본 가중치 초기값 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

## 6.1.8 MNIST 데이터셋으로 본 갱신 방법 비교

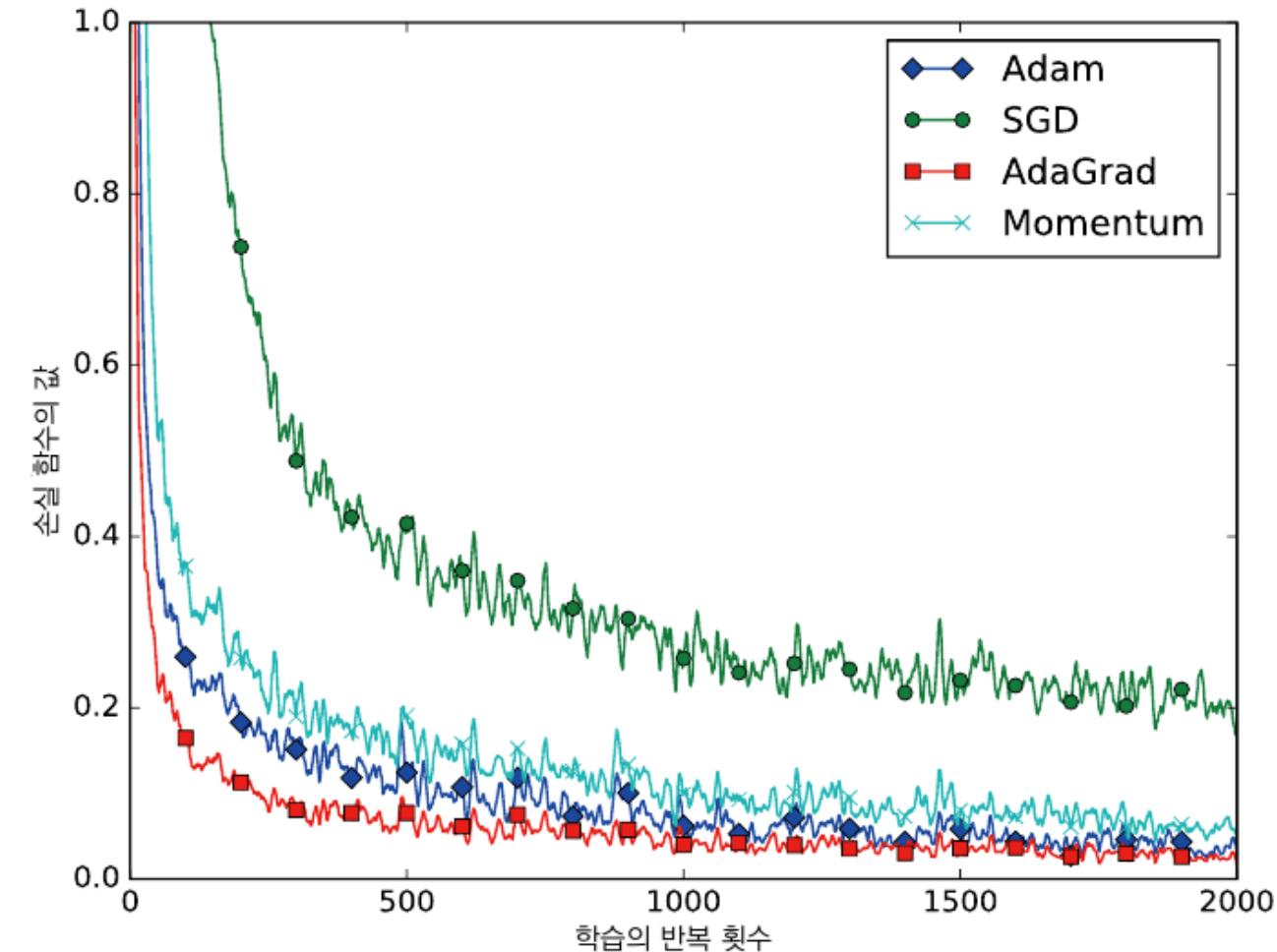
복습

```
# MNIST 손글씨 숫자 인식을 대상으로 4 방법 탐색의 수렴 과정 비교
```

각 층이 100개 뉴런으로 구성된 5층 신경망에서 ReLU 활성화 함수 사용.

학습진도는 SGD가 가장 느리고 AdaGrad가 가장 빠름. 보통, SGD보다 다른 세 기법이 빠르게 학습하고, 정확도도 살짝 높다.

그림 6-9 MNIST 데이터셋에 대한 학습 진도 비교



ch06/optimizer\_compare\_mnist.py

## 6.2 가중치의 초기값

가중치의 초기값을 무엇으로 설정하느냐가 신경망 학습의 성패를 가르는 일이 종종 있음.

권장 초기값에 대해 설명하고,

실험을 통해 실제로 신경망 학습이 신속하게 이뤄지는 모습을 확인함.

## 6.2.1 초기값을 0으로 하면?

이제부터 오버피팅을 억제해 범용 성능을 높이는 테크닉인 **가중치 감소**<sub>weight decay</sub> 기법을 소개  
가중치 감소는 간단히 말하자면 가중치 값이 작아지도록 학습하는 방법.  
가중치 값을 작게 하여, 오버피팅이 일어나지 않게 하는 것이다.

가중치를 작게 할 수 있는 한가지 방법은... “작은 값에서 시작하는 것이다.”  
“초깃값(initial value)을 작게 두자” 얼마만큼? 어떻게 작게? 0에 가깝게?  
그렇다고,  
모두 0으로 두면 안 된다. 왜? 모두 같은 값이라 학습이 안 된다.

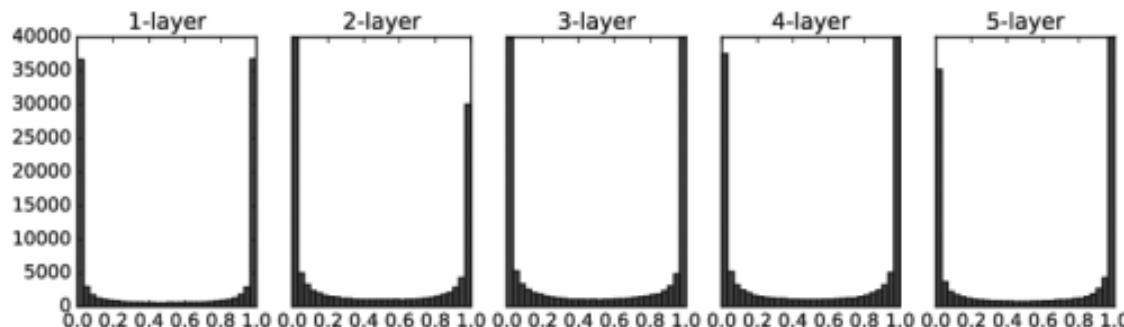
예전에,  $0.01 \times np.random.randn(10,100)$ 라고 작게 다른 값들로 둔 적이 있다.

## 6.2.2 은닉층의 활성화값 분포

은닉층의 활성화값(활성화 함수의 출력 데이터)의 분포를 관찰하면 중요한 정보를 얻을 수 있다.

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

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



데이터가 0과 1에 치우쳐 분포하게 되면 역전파의 기울기 값이 점점 작아지다가 사라진다.

이것이 기울기 소실 gradient vanishing이라 알려진 문제이다

## 6.2.2 은닉층의 활성화값 분포

은닉층의 활성화값(활성화 함수의 출력 데이터)의 분포를 관찰하면 중요한 정보를 얻을 수 있다.

```
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]

        # 초기값을 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)

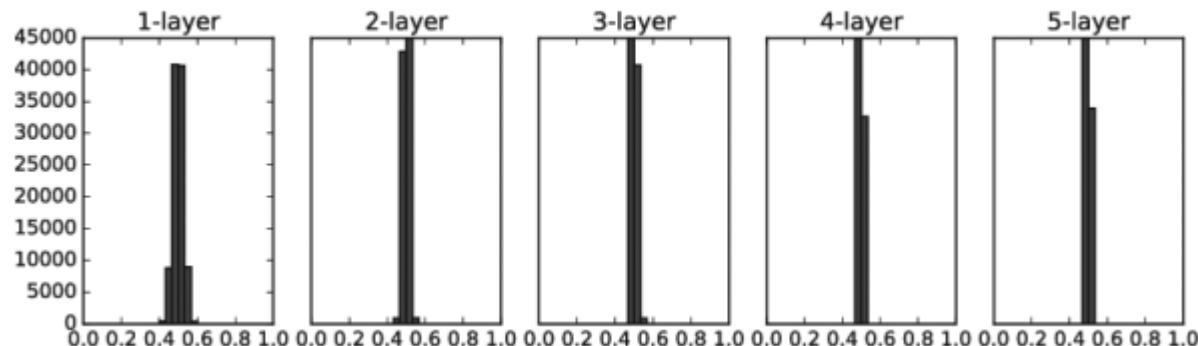
        # 활성화 함수도 바꿔가며 실험해보자 !
        z = sigmoid(a)
        # z = ReLU(a)
        # z = tanh(a)

        activations[i] = z
```

## 6.2.2 은닉층의 활성화값 분포

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

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



뉴런 100개가 거의 같은 값을 출력한다 > 뉴런 1개와 별반 다를 것이 없다.

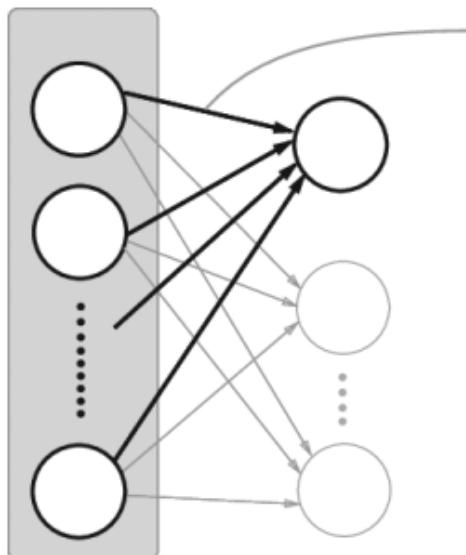
활성화 값이 치우치면 표현력을 제한하여 성능이 감소한다.

(한 값으로 몰리면)

## 6.2.2 은닉층의 활성화값 분포

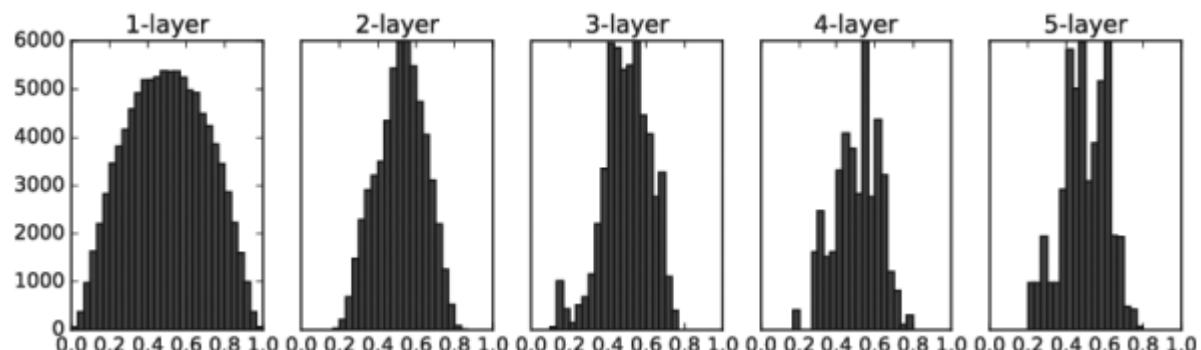
### Xavier 사비에르 초기값

그림 6-12 Xavier 초기값 : 초기값의 표준편차가  $\frac{1}{\sqrt{n}}$  이 되도록 설정 ( $n$ 은 앞 층의 노드 수)  
 $n$ 개의 노드



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

그림 6-13 가중치의 초기값으로 'Xavier 초기값'을 이용할 때의 각 층의 활성화값 분포



Xavier 초기값의 결과는 층이 깊어지면서 형태가 다소 일그러지지만, 앞의 방식보다는 확실히 넓게 분포된다. 데이터가 적당히 퍼져 있으므로 학습이 효율적으로 이뤄질 것으로 기대된다.

### 6.2.3 ReLU를 사용할 때의 가중치 초기값

Xavier 초기값은 활성화 함수가 선형인 것을 전제로 이끈 결과이다. Sigmoid 함수와 tanh 함수는 원점 대칭이고 중앙 부근이 선형인 함수로 볼 수 있다

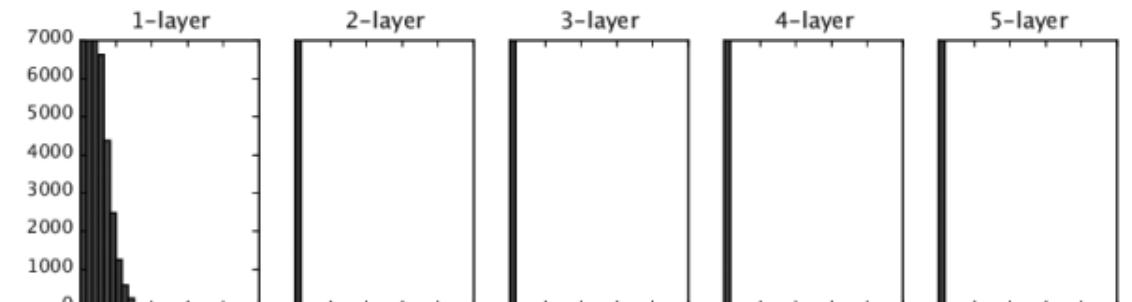
반면, Relu는 카이밍 히 Kaiming He의 이름을 딴 **He 초기값**을 사용할 것을 권합니다.

He 초기값은 앞 계층의 노드가  $n$ 개일 때, 표준편차가  $\sqrt{\frac{2}{n}}$ 인 표준 정규분포를 사용한다. (Xavier 초기값이  $\sqrt{\frac{1}{n}}$ 이었다)

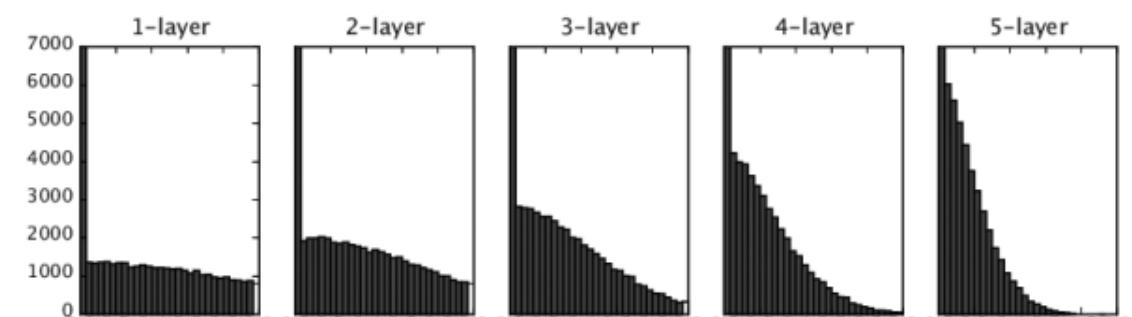
왜냐하면, ReLU는 음의 영역이 0이라서 더 넓게 분포시키기 위해 2배의 계수가 필요하다고 해석할 수 있겠습니다.

### 6.2.3 ReLU를 사용할 때의 가중치 초기값

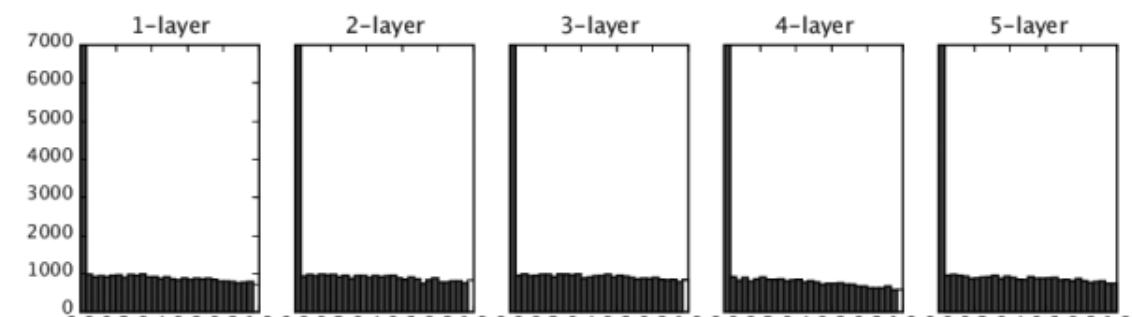
그림 6-14 활성화 함수로 ReLU를 사용한 경우의 가중치 초기값에 따른 활성화값 분포 변화



표준편차가 0.01인 정규분포를 가중치 초기값으로 사용한 경우



Xavier 초기값을 사용한 경우



He 초기값을 사용한 경우

이상의 실험 결과들을 바탕으로,

활성화 함수로 ReLU를 사용할 때는 He 초기값을,

Sigmoid나 tanh를 사용할 때는 Xavier 초기값을 사용하겠습니다.

이것이 모범 사례라고 할 수 있습니다.

## 6.2.4 MNIST 데이터셋으로 본 가중치 초기값 비교

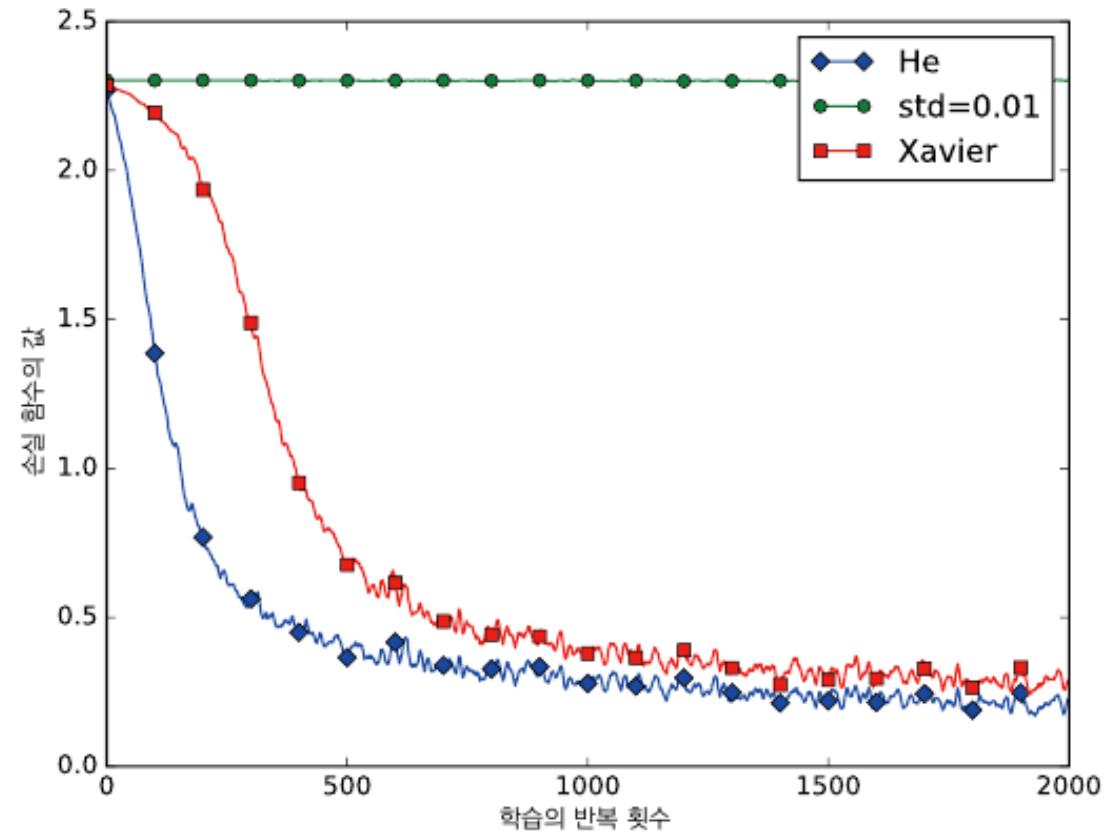
5층 신경망(층별 뉴런 수 100개), 활성화 함수로 ReLU를 사용.

ch06/weight\_init\_compare.py

가중치 초기값은 신경망 학습에 매우 중요하다.

가중치 초기값에 따라 신경망 학습의 성패가  
갈리는 경우가 많다.

그림 6-15 MNIST 데이터셋으로 살펴본 '가중치의 초기값'에 따른 비교

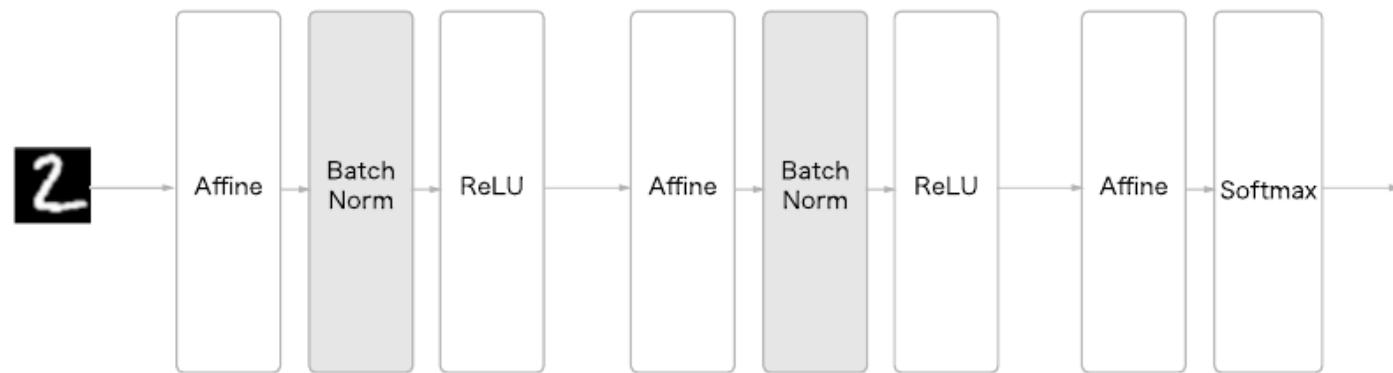


### 6.3.1 배치 정규화 알고리즘

각 층이 활성화를 적당히 퍼뜨리도록 강제하는 방법

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

그림 6-16 배치 정규화를 사용한 신경망의 예



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

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

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

[식 6.7]

$$y_i \leftarrow \gamma \hat{x}_i + \beta \quad [식 6.8]$$

배치 정규화는 Two step. [식 6.7]과 [식 6.8]을 수행한다.  
 $\gamma$ 감마와  $\beta$ 베타는 매개변수로 데이터를 이용하여 학습한다.

## 6.3.2 배치 정규화의 효과

ch06/batch\_norm\_test.py

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

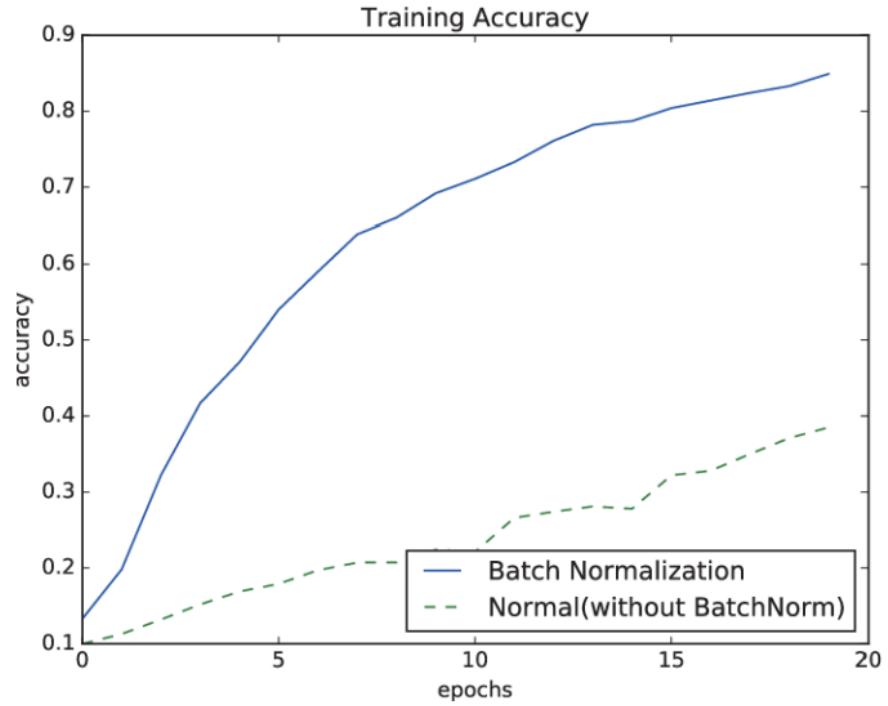
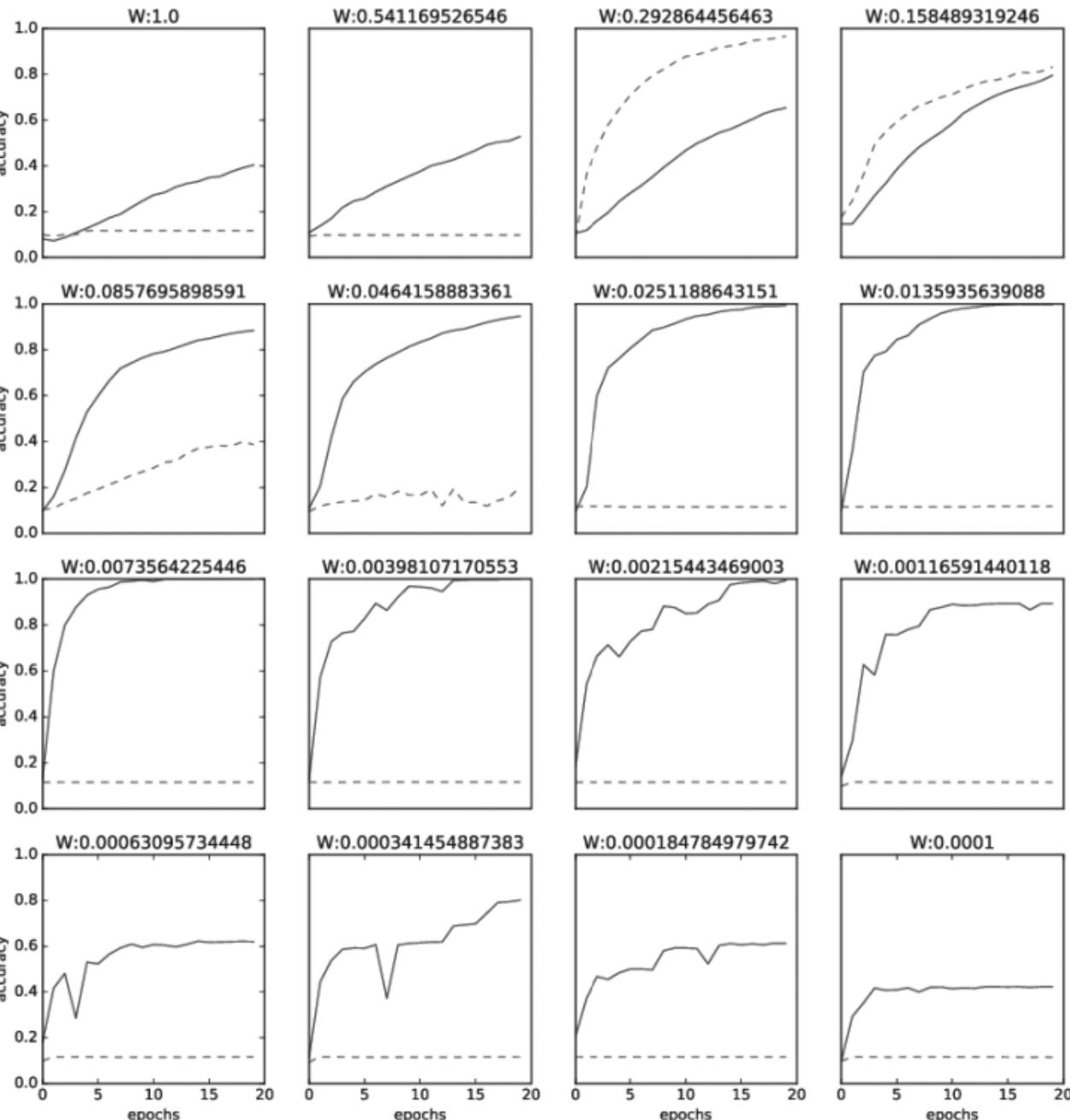


그림 6-19 실선이 배치 정규화를 사용한 경우, 점선이 사용하지 않은 경우 : 가중치 초기값의 표준편차는 각 그래프 위에 표기



## 6.3.2 배치 정규화의 효과

ch06/batch\_norm\_test.py

```
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
```

## 6.3.2 배치 정규화의 효과

ch06/batch\_norm\_test.py

```
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
```

## 6.3.2 배치 정규화의 효과

ch06/batch\_norm\_test.py

# [그림 6-19] 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()
```

## 6.4 바른 학습을 위해

기계학습에서 오버피팅(Overfitting)이 문제가 되는 경우가 많다.

오버피팅이란 신경망이 훈련 데이터에만 지나치게 적응되어 그 외의 데이터에는 제대로 대응하지 못하는 상태를 말합니다. 훈련 데이터에는 포함되지 않은, 아직 보지 못한 데이터도 바르게 식별해 내는 모델이 바람직하다.

6.4 절에서는 오버피팅을 억제하는 기술인 가중치 감소, 드롭아웃에 대해 배운다.

다음 시간에~