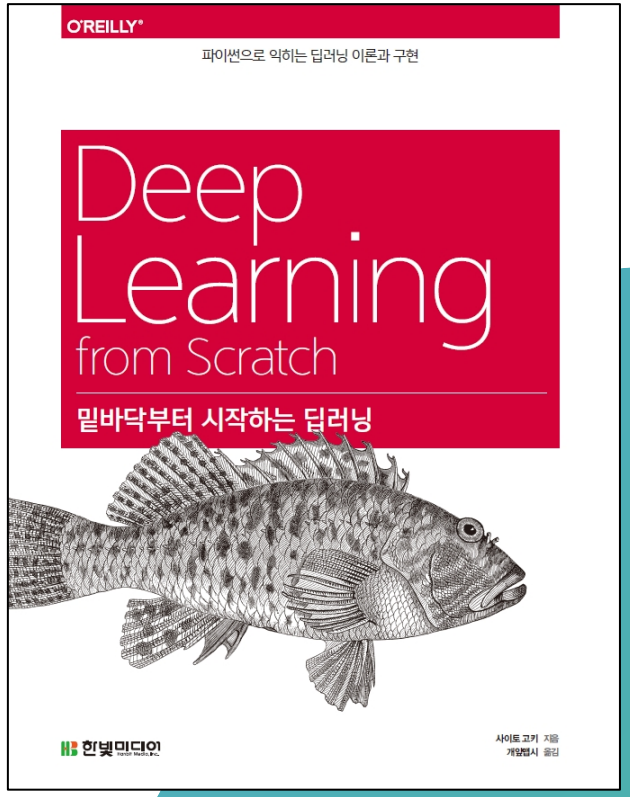
CHAPTER 6 학습 관련 기술들

밑바닥부터 시작하는 딥러닝



홍익대학교 컴퓨터공학과
박 준

이 책의 학습 목표

- CHAPTER 1 파이썬에 대해 간략하게 살펴보고 사용법 익히기
- CHAPTER 2 퍼셉트론에 대해 알아보고 퍼셉트론을 써서 간단한 문제를 풀어보기
- CHAPTER 3 신경망의 개요, 입력 데이터가 무엇인지 신경망이 식별하는 처리 과정 알아보기
- CHAPTER 4 손실 함수의 값을 가급적 작게 만드는 경사법에 대해 알아보기
- CHAPTER 5 가중치 매개변수의 기울기를 효율적으로 계산하는 오차역전파법 배우기
- CHAPTER 6 신경망(딥러닝) 학습의 효율과 정확도를 높이기
- CHAPTER 7 CNN의 메커니즘을 자세히 설명하고 파이썬으로 구현하기
- CHAPTER 8 딥러닝의 특징과 과제, 가능성, 오늘날의 첨단 딥러닝에 대해 알아보기

Contents

Contents

CHAPTER 6 학습 관련 기술들

신경망(딥러닝) 학습의 효율과 정확도를 높이기

6.1.2 확률적 경사 하강법(SGD)

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

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

Update(params, grads) 메서드는 SGD 과정에서 반복해서 불린다

```
class SGD:
    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]
```

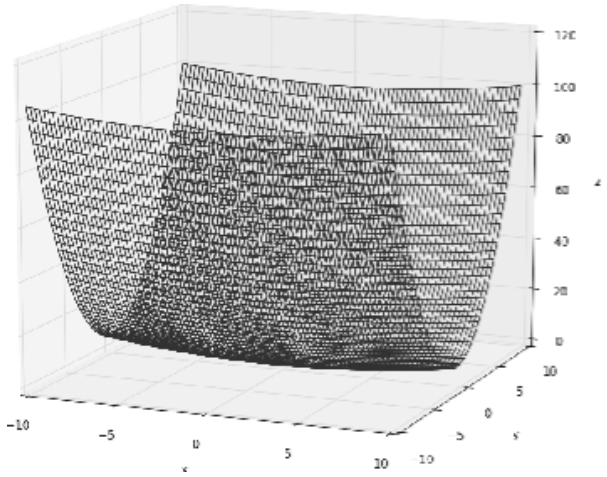
Optimizer class를 분리해서 코드를 작성
—> 기능 모듈화

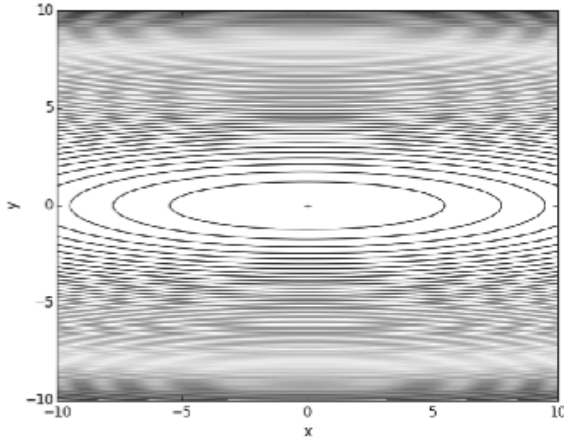
이 경우에는 stochastic gradient descent

6.1.3 SGD의 단점

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

이 함수는 [그림 6-1]의 왼쪽과 같이 '밥그릇'을 x축 방향으로 늘인 듯한 모습이고, 실제로 그등고선은 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 y축 방향은 크고x축 방향은 작다는 것이 특징
[그림 6-2] 최소가 되는 점 (0,0). 기울기 대부분이 (0,0)을 향하지 않음

[그림 6-3] 지그재그로 이동: 비효율적

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

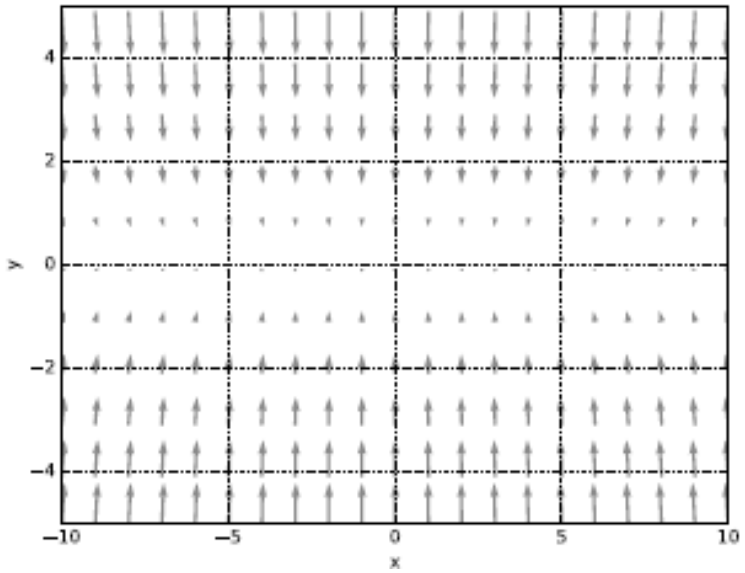
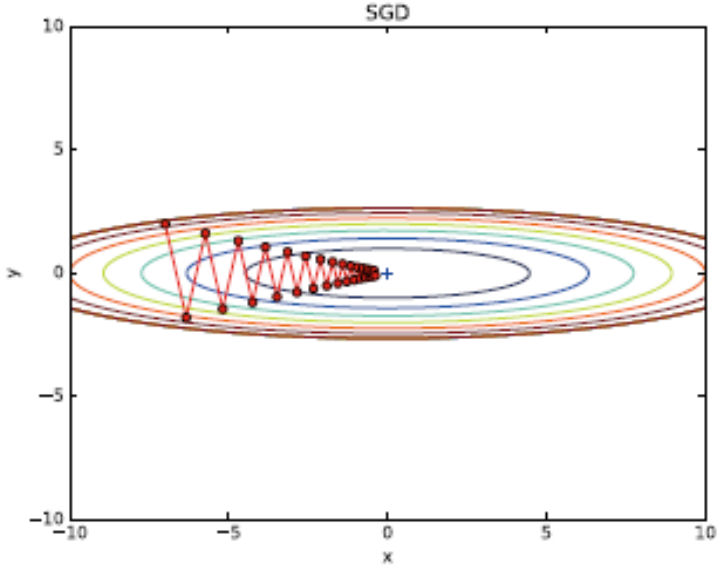


그림 6-3 SGD에 의한 최적화 갱신 경로 : 최솟값인 (0, 0)까지 지그재그로 이동하니 비효율적이다.



해결책:
feature scaling
Momentum, AdaGrad, Adam

SECTION 06 학습 관련 기술들

6.1.4 모멘텀

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

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

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

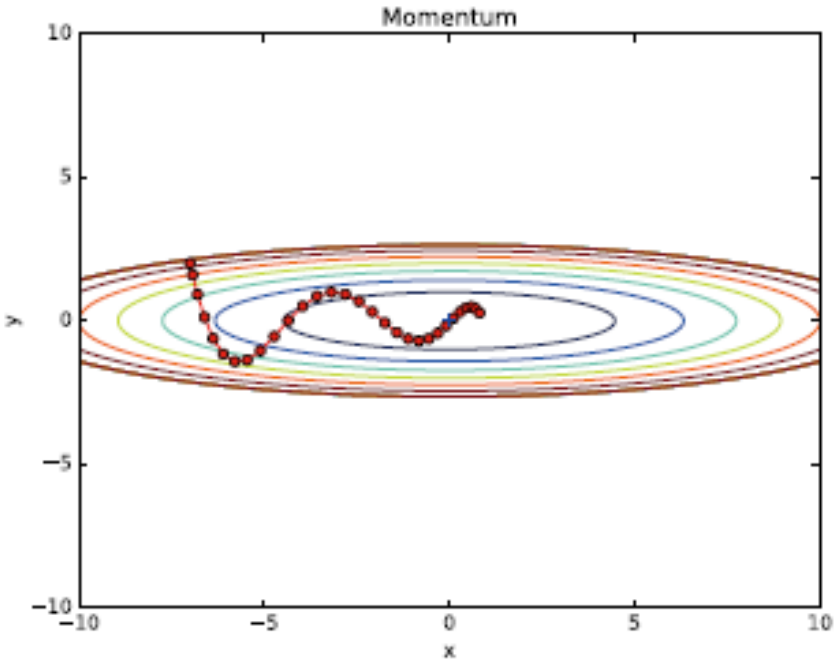
```
class Momentum:
    def __init__(self, lr=0.01, momentum=0.9):
        self.lr = lr
        self.momentum = momentum
        self.v = None

    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]
```

Momentum: 물리의 운동량
[식 6.3] v: 속도
공이 바닥에서 구르는 것과 같은 움직임

또, [식 6.3]의 $\alpha\mathbf{v}$ 항은 물체가 아무런 힘을 받지 않을 때 서서히 하강시키는 역할: : 지면 마찰이나 공기 저항과 유사
(α는 0.9 등의 값으로 설정)

[그림 6.5] 지그재그 정도가 감소
x 방향 힘은 작아도 방향이 변하지 않아 누적 효과가 커짐
y 방향은 위아래 방향이 상충하여 속도가 안정적이지 않음

그림 6-5 모멘텀에 의한 최적화 갱신 경로



6.1.5 AdaGrad

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

AdaGrad: 각각의 매개변수에 맞춰 학습률 조정
h: 기존 기울기 제곱을 계속 더해 줌 (연산: 행렬 원소별 제곱)
많이 움직인 파러미터의 학습률 낮춤

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

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

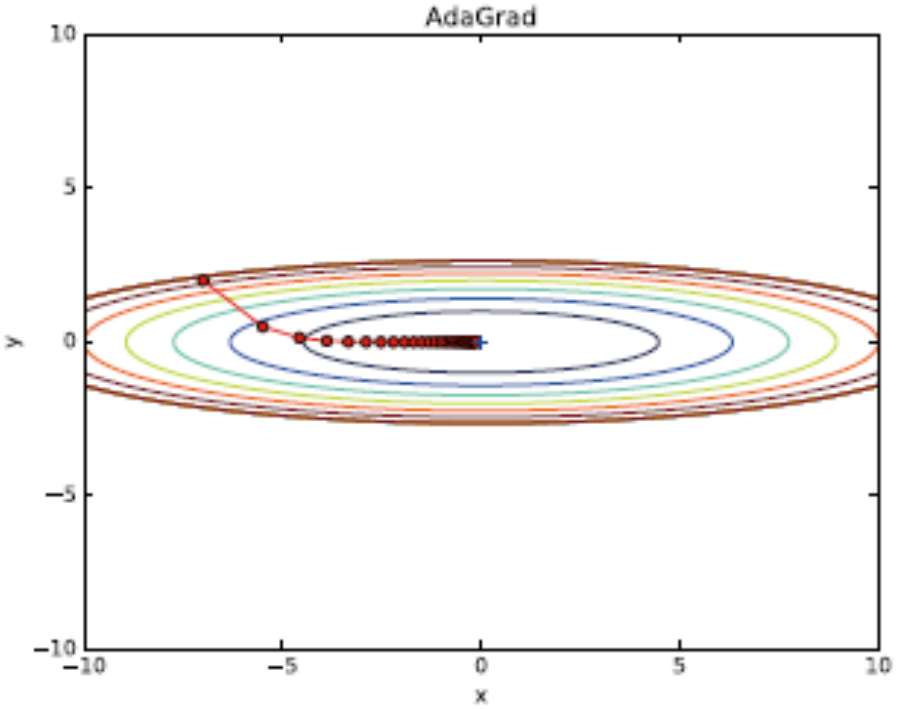
h 과거의 기울기 제곱의 누적 합
—> 학습을 진행할수록 갱신 강도가 약해짐
문제점: 어느 순간 0에 가까운 값이 되어 갱신이 되지 않음

—> 보완방법: RMSProp
오래된 기울기는 서서히 잊어버리게 설계 (지수이동평균)

6.1.5 AdaGrad

```
class 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)
```

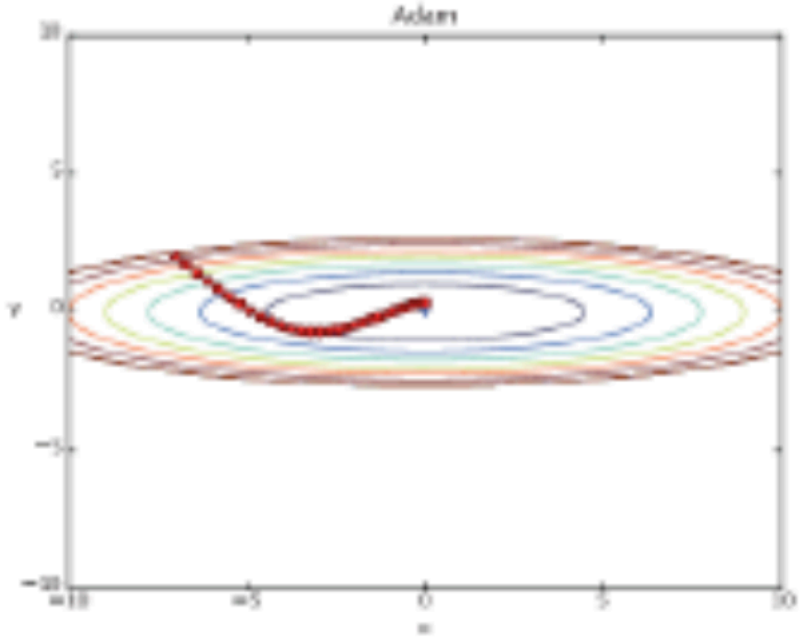
그림 6-6 AdaGrad에 의한 최적화 갱신 경로



6.1.6 Adam

Momentum + AdaGrad

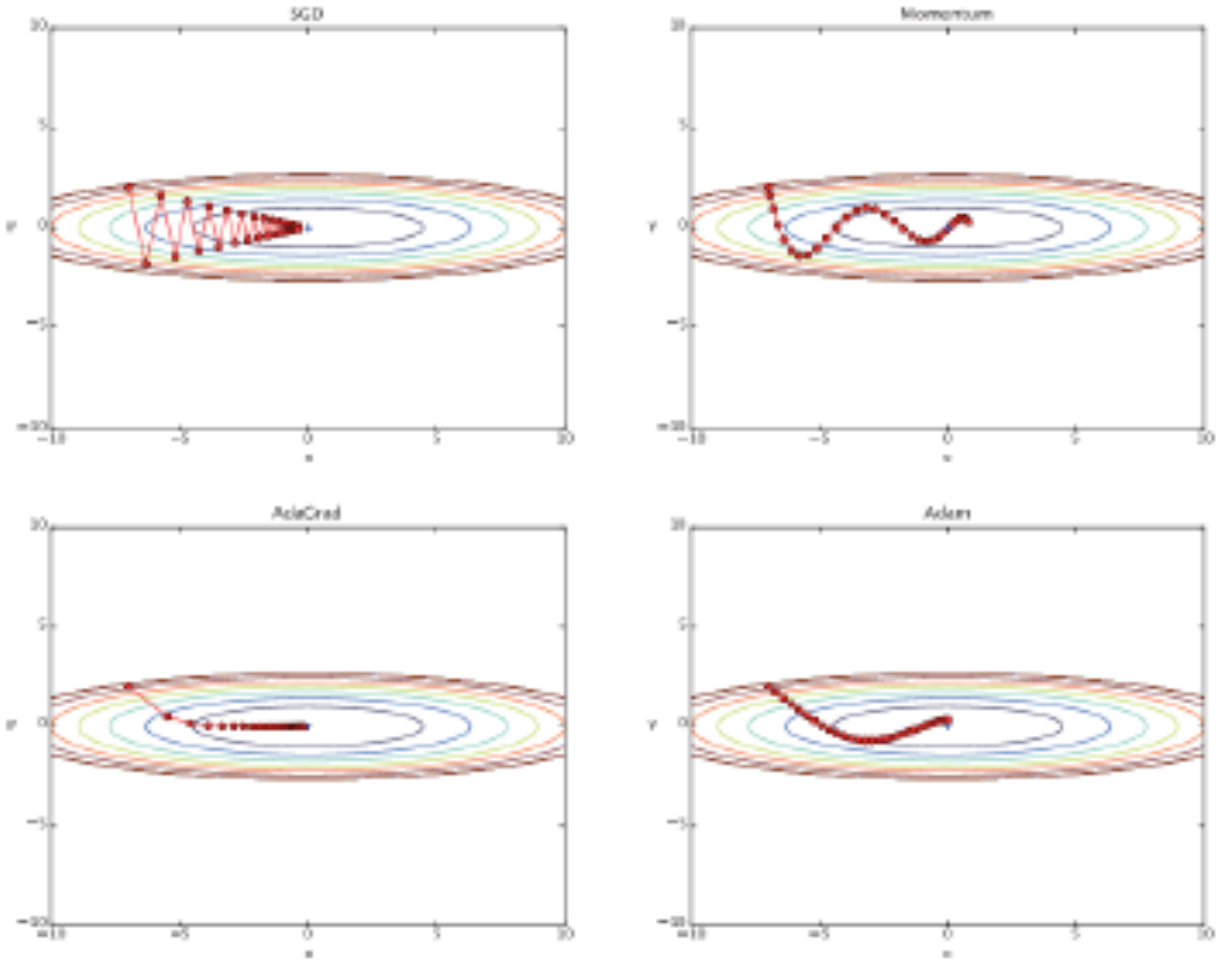
두 가지 방법의 장점 조합



6.1.7 어느 갱신 방법을 이용할 것인가?

이들 네 기법의 결과를 비교

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

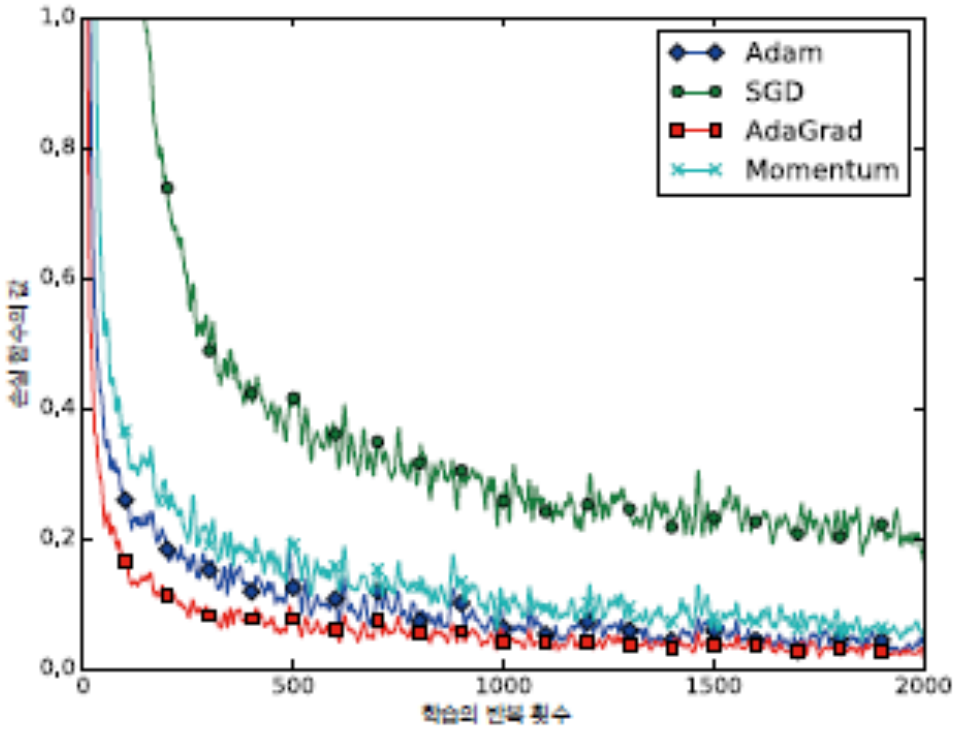


그림에서는 AdaGrad가 좋아보이지만 경우에 따라 달라짐
- 데이터
- 다른 하이퍼파라미터 등

많은 연구에서 SGD 아직도 많이 사용
요즘은 Adam의 성능이 좋은 사례가 많음

6.1.8 MNIST 데이터셋으로 본 갱신 방법 비교

그림 6-9 MNIST 데이터셋에 대한 학습 진도 비교



일반적으로 SGD보다 다른 세 기법의 학습도 빠르고
최종 정확도도 높은 경우가 많음

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: Regularization

가중치 값을 작게 하여 오버피팅이 일어나지 않게 하는 것이다.

가중치 값을 작게 하기 위해 작은 값으로 초기화
0으로 초기화하면?

- 학습이 이루어지지 않음
- back-propagation에서 모든 가중치 값이 똑같이 갱신되기 때문 (중간층 activation 값이 forward propagation 때 같은 값으로 계산되므로)

6.2.2 은닉층의 활성화값 분포

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

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

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

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

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

    w = np.random.randn(node_num, node_num) * 1
    a = np.dot(x, w)
    z = sigmoid(a)
    activations[i] = z
```

사례: 5층 신경망(100 x 100 x 100 x 100 x 100 x 100), 시그모이드 활성함수

```
# 히스토그램 그리기
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    plt.hist(a.flatten(), 30, range=(0,1))
plt.show()
```

이 코드를 실행하면 [그림 6-10]의 히스토그램을 얻을 수 있다

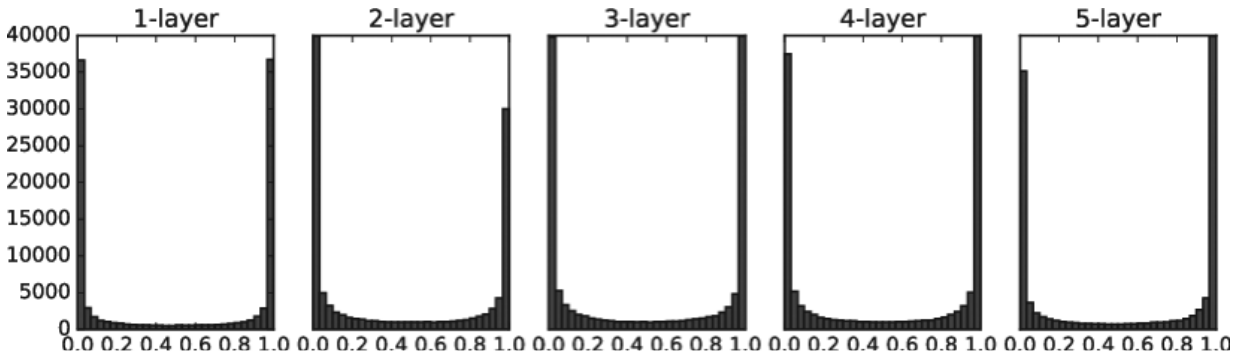


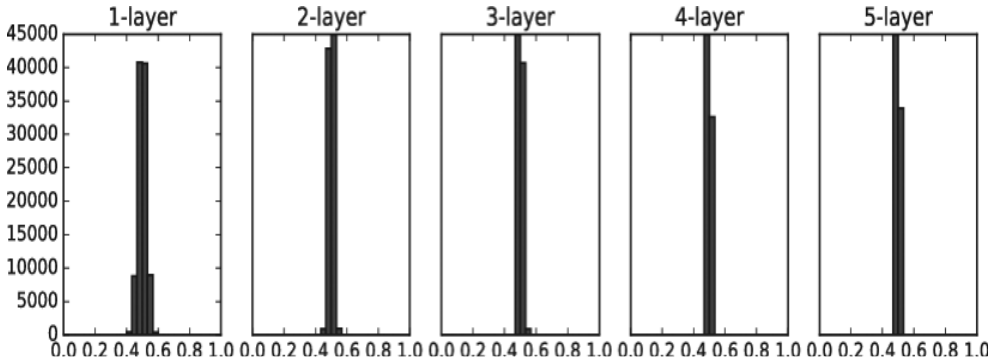
그림 6-10 가중치를 표준편차가 1인 정규분포로 초기화할 때의 각 층의 활성화값 분포

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: 0.5 근처에 집중

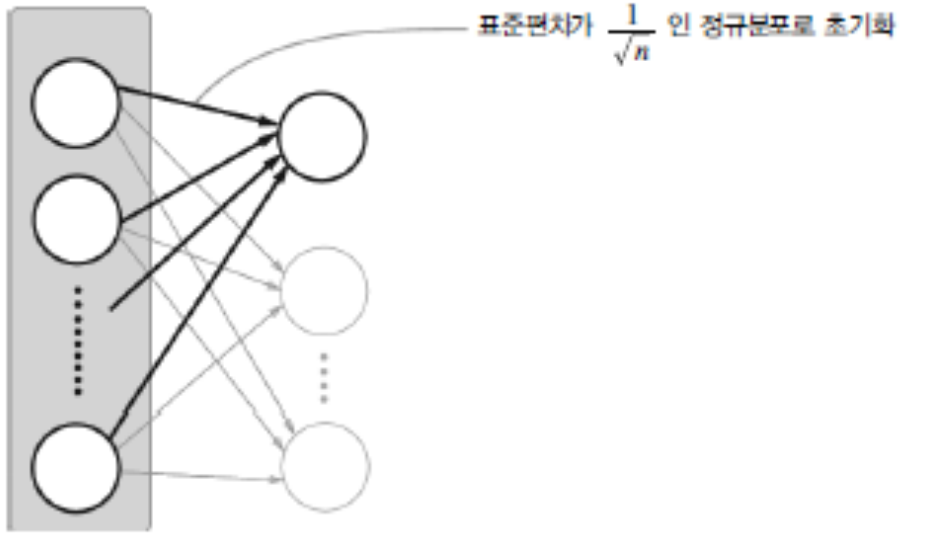


활성화 값이 0.5 근처에 집중
활성화값들이 치우치면 표현력을 제한한다는 문제
—> 활성화 값을 고르게 분포시킬 필요

6.2.2 은닉층의 활성화값 분포

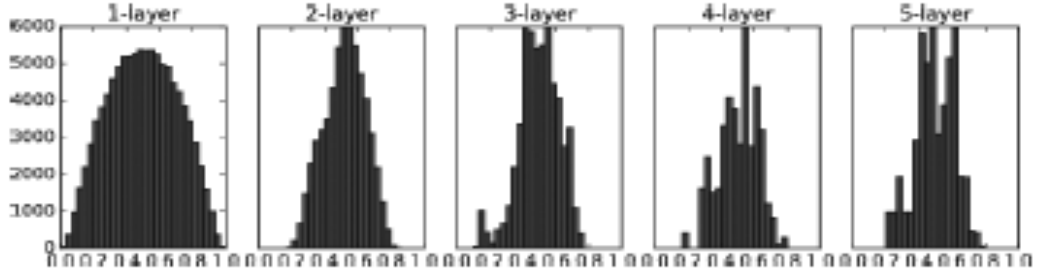
가중치 초깃값인, 일명 **Xavier** 초깃값을 사용 (Bengio 논문에기초)
: 표준으로 많이 사용

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



앞에서 보다는 고르게 분포
층이 깊어지면서 다소 일그러짐
일그러짐은 Sigmoid 대신 tanh 사용하면 개선됨

6.2.3 ReLU를 사용할 때의 가중치 초깃값

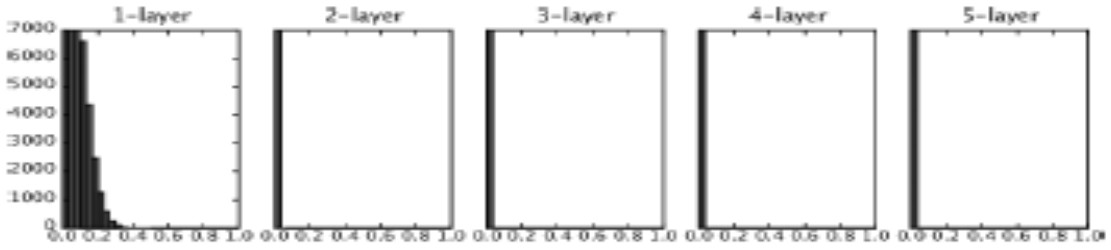
Xavier 초깃값은 활성화 함수가 선형인 것을 전제로 이끈 결과이다. Sigmoid 함수와 tanh 함수는 좌우 대칭이라 중앙 부근이 선형인 함수로 볼 수 있다

ReLU 사용시, ReLU에 특화된 초기값 필요
이 특화된초깃값을 찾아낸 카이밍 히Kaiming He의 이름을 따 **He** 초깃값[10]이라 한다

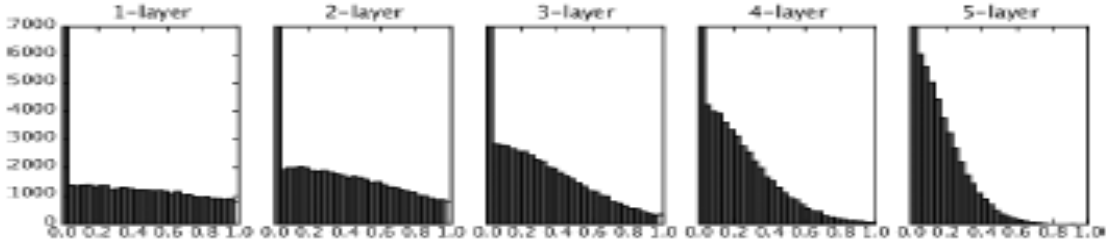
표준편차가 $\sqrt{\frac{2}{n}}$ 인 정규분포 사용

ReLU의 경우 음수 부분이 0이므로 더 넓은 분포 사용

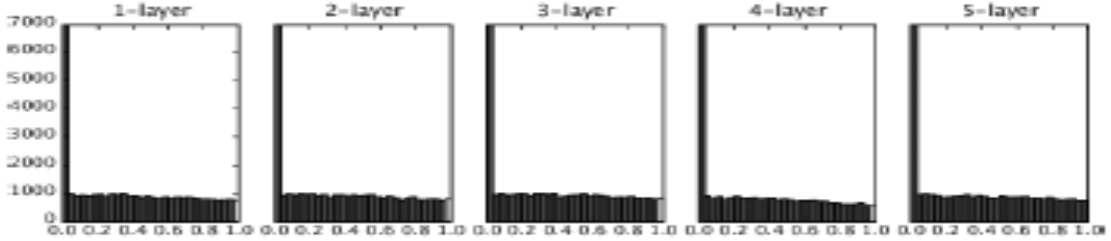
그림 6-14 활성화 함수로 ReLU를 사용한 경우의 가중치 초깃값에 따른 활성화값 분포 변화



표준편차가 0.01인 정규분포를 가중치 초깃값으로 사용한 경우



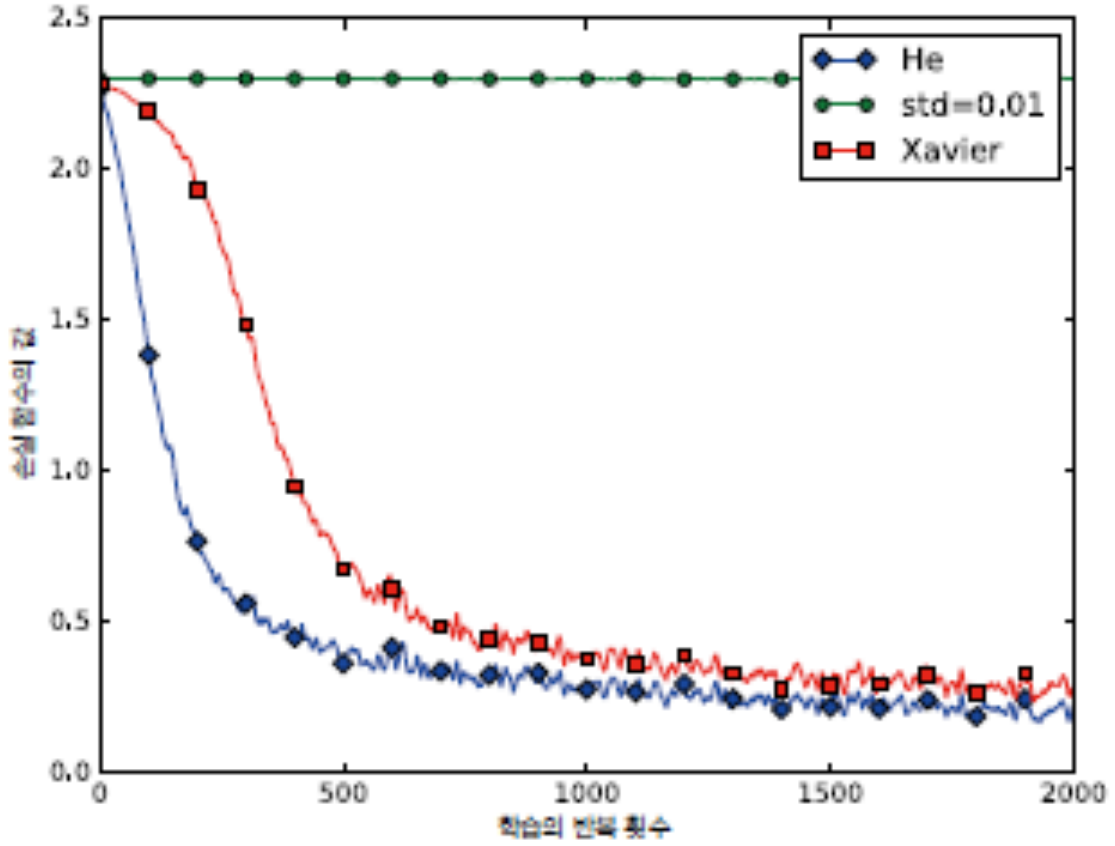
Xavier 초깃값을 사용한 경우



He 초깃값을 사용한 경우

6.2.4 MNIST 데이터셋으로 본 가중치 초깃값 비교

그림 6-15 MNIST 데이터셋으로 살펴본 '가중치의 초깃값에 따른 비교



5층 신경망,
각 층의 노드: 100
ReLU

std = 0.01 : 너무 작아 학습이 이루어 지지 않음

6.3.1 배치 정규화 알고리즘

- 앞의 예: 적절한 초기값 —> 활성화값을 적절하게 퍼뜨림

- 각 층의 활성화 값을 적절하기 퍼뜨리도록 강제화
- —> batch normalization (2015)

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

- 데이터 분포를 정규화하는 batch norm 계층 삽입

그림 6-16 배치 정규화를 사용한 신경망의 예

- 학습시 미니배치 단위로 정규화
  - 평균 0, 분산 1이 되도록
  - 식 6.7
- 데이터 분포가 덜 치우치게 하는 효과

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

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

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

[식 6.7]

6.3.1 배치 정규화 알고리즘

- 배치 정규화 계층마다 확대(Scale), 이동(Shift) 변환 수행
- 초기값 1, 0으로 시작, 학습하면서 조정

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

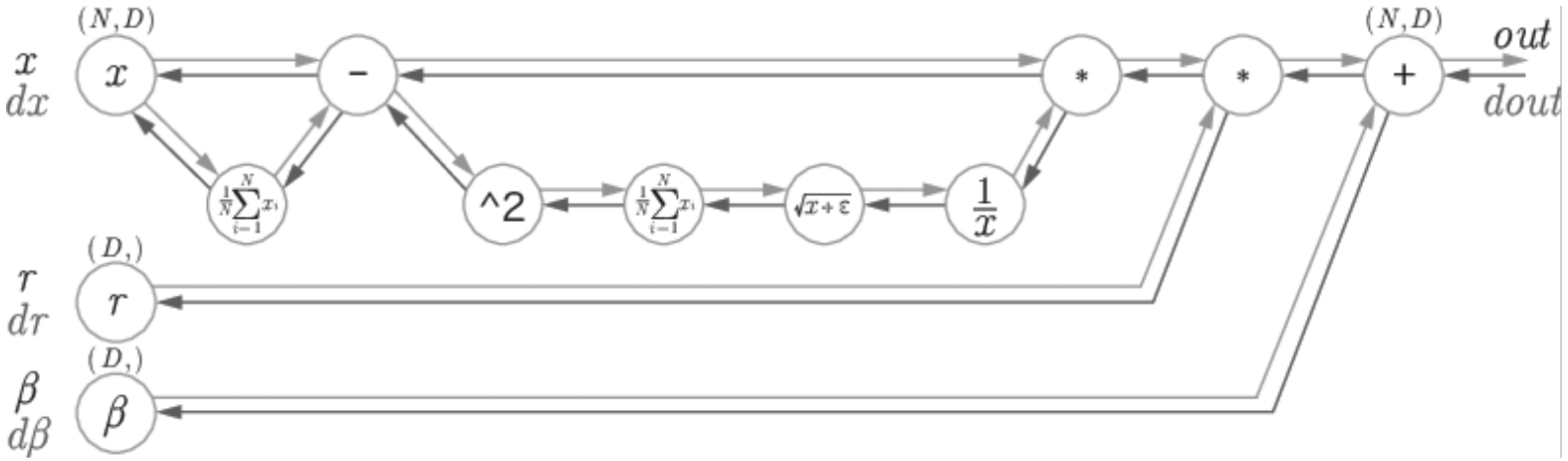


그림 6-17 배치 정규화의 계산 그래프[13]

6.3.2 배치 정규화의 효과

그림 **6-18** 배치 정규화의 효과 :

배치 정규화가 학습 속도를 높인다.
초기값에 의존하지 않는다.

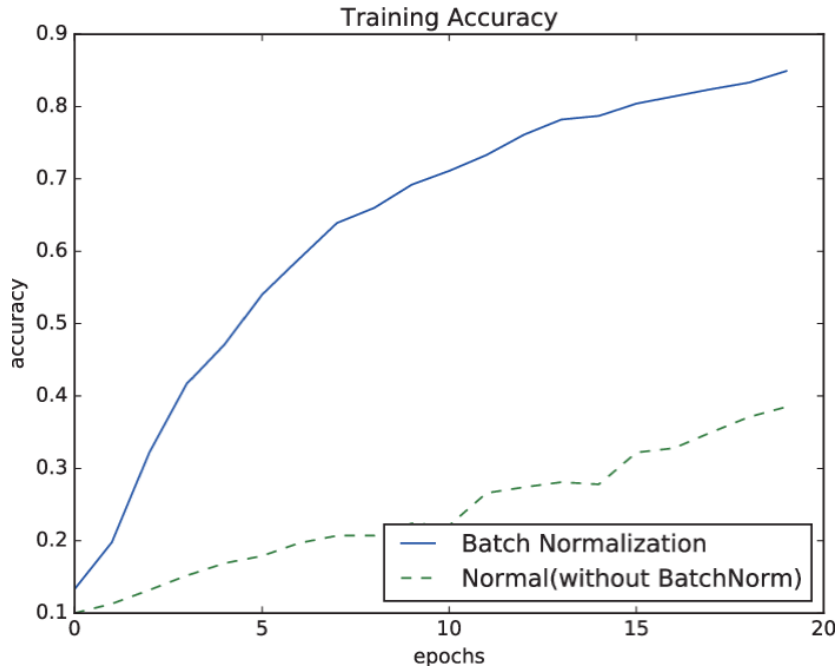
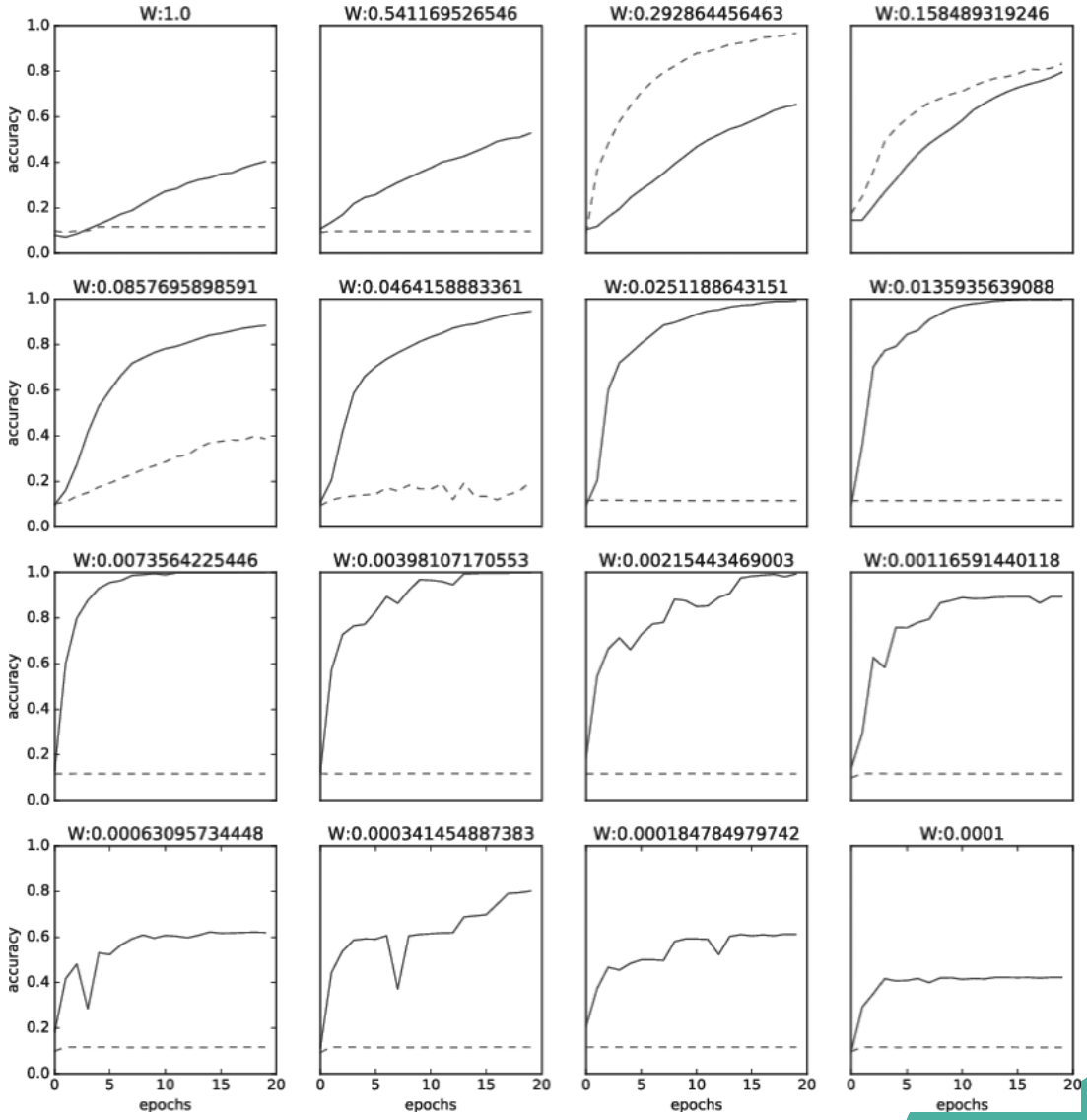


그림 **6-19** 실선이 배치 정규화를 사용한 경우, 점선이 사용하지 않은 경우 :
다양한 가중치 초기값 표준편차



6.4.1 오버피팅

..매개변수가 많고 표현력이 높은 모델
..훈련 데이터가 적용

일부러 오버핏팅 일으킨 경우
7층 네트워크, 300개의 데이터

Train : Test 정확도 차이가 크다

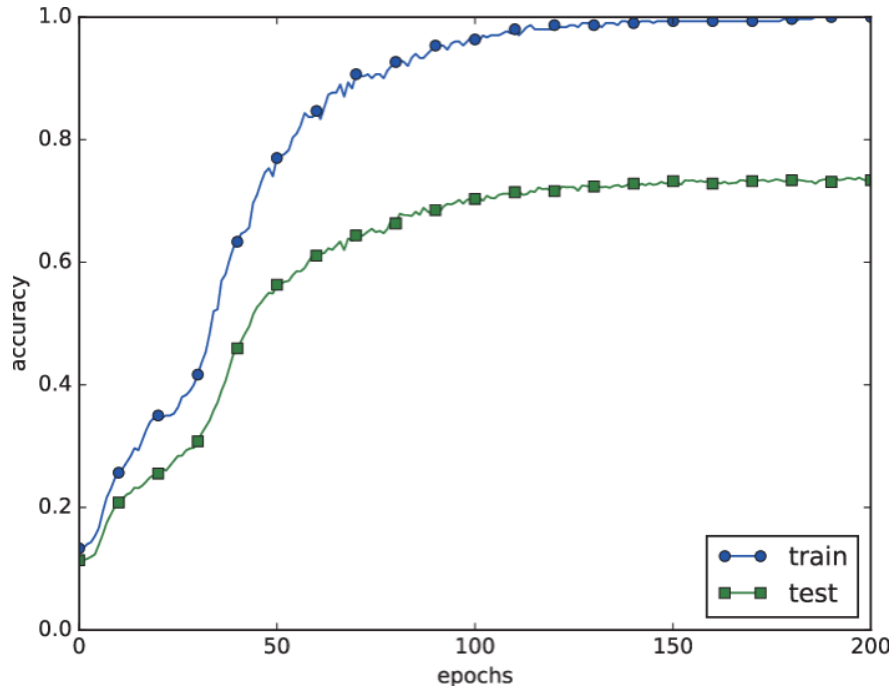


그림 6-20 훈련 데이터(train)와 시험 데이터(test)의 에폭별 정확도 추이

```
(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]
```

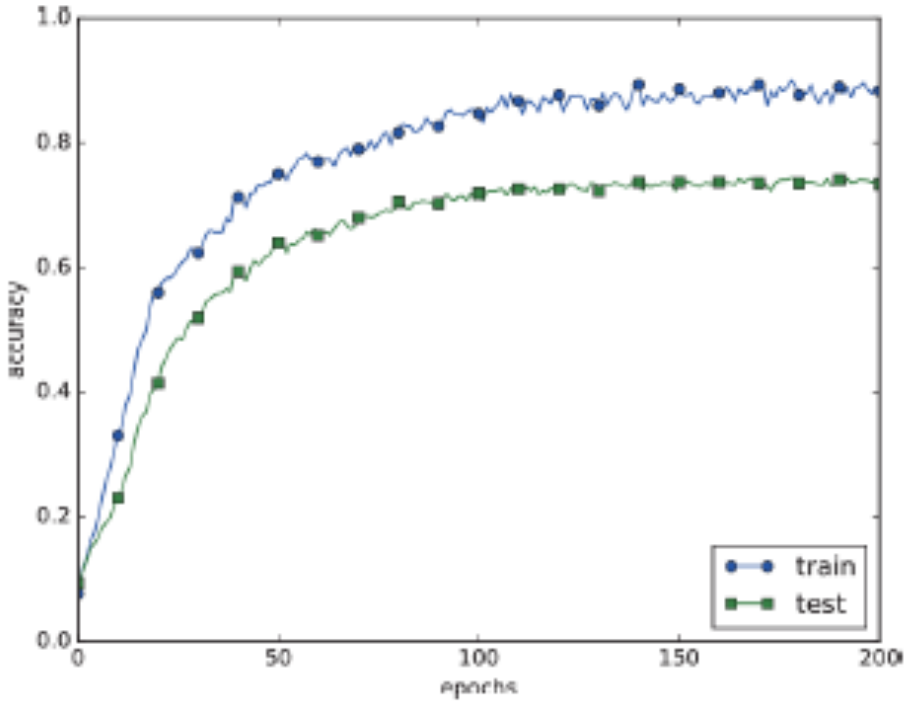
```
network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100,
100, 100], output_size=10)
optimizer = SGD(lr=0.01) # 학습률이 0.01인 SGD로 매개변수 갱신

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

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)

        epoch_cnt += 1
        if epoch_cnt >= max_epochs:
            break
```

6.4.2 가중치 감소

오버피팅 억제용으로 예로부터 많이 이용해온 방법 중 가중치 감소weight decay사용:
L2 Regularization
오버피팅은 가중치 값이 커서 발생하는 경우가 많으므로

차이가 다소 감소
Training accuracy가 1보다 꽤 작음

그림 6-21 가중치 감소를 이용한 훈련 데이터(train)와 시험 데이터(test)에 대한 정확도



6.4.3 드롭아웃

신경망이 복잡해지면 weigh decay만으로 해결 어려움
Drop out: 학습 시 임의로 뉴런 삭제

무작위로 삭제할 뉴런 선택
시험 시에는 모든 뉴런에 신호 전달
(옵션) 시험 때 각 뉴런의 출력에 훈련 시 삭제 비율 곱하여 출력

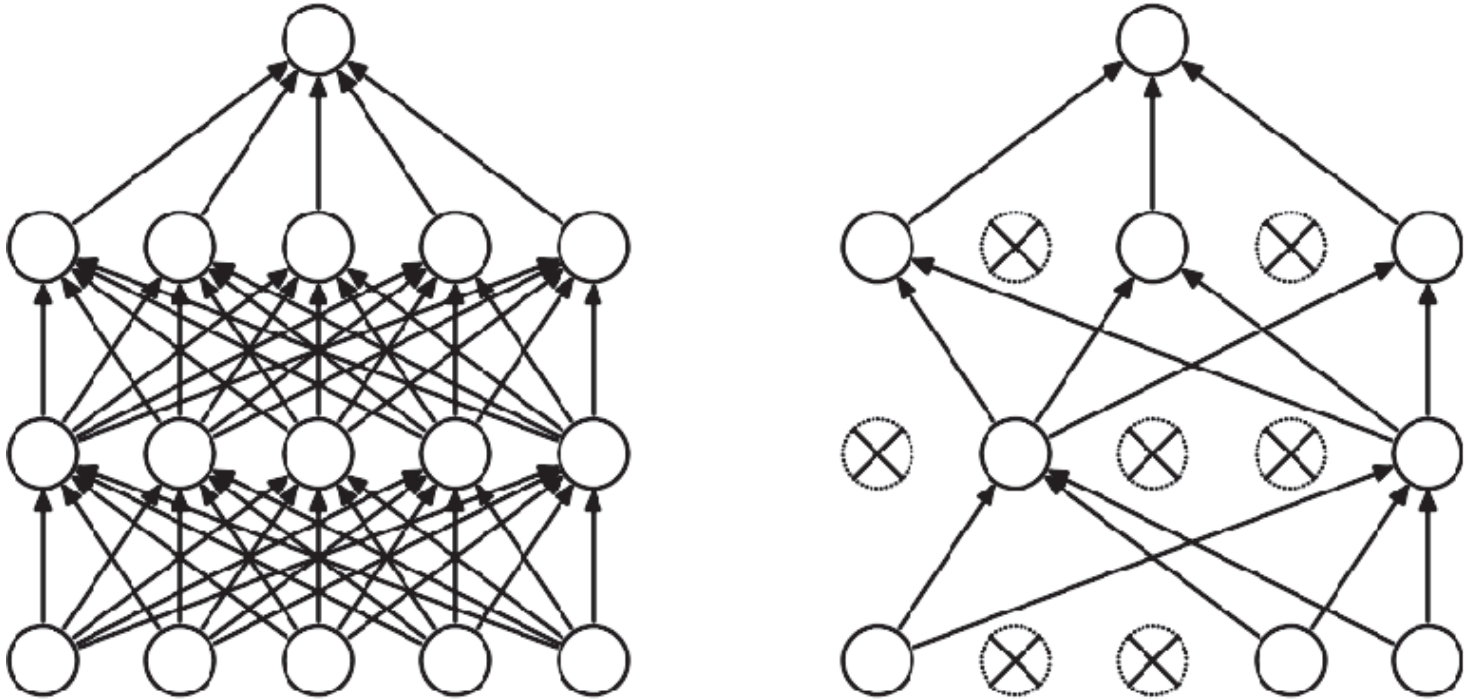
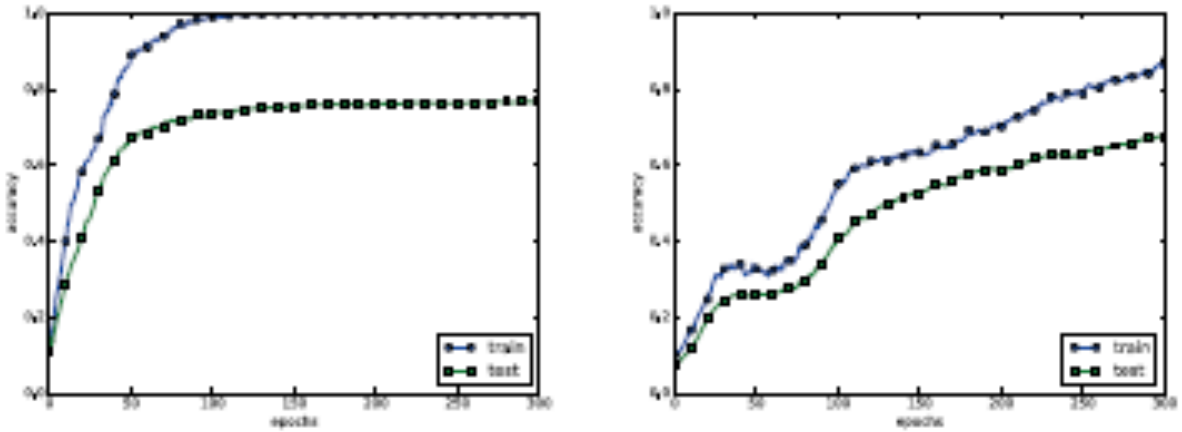


그림 **6-22** 드롭아웃의 개념(문헌[14]에서 인용) : 왼쪽이 일반적인 신경망, 오른쪽이 드롭아웃을 적용한 신경망. 드롭아웃은 뉴런을 무작위로 선택해 삭제하여 신호 전달을 차단한다.

6.4.3 드롭아웃

```
class Dropout:
    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
            return x * self.mask
        else:
            return x * (1.0 - self.dropout_ratio)

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

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



드럽아웃은 앙상블과 비슷한 효과
앙상블: 비슷한 구조의 네트워크 여러개 학습 시킨 후 평균을 내어 출력

6.5.1 검증 데이터

하이퍼파라미터를 조정할 때는 하이퍼파라미터 전용 확인 데이터가 필요하다.
: 각 층의 뉴런 수, 배치 크기, 학습률, weight decay 등

하이퍼파라미터 조정용 데이터를 일반적으로 검증 데이터validation data라고 부른다
검증 데이터를 사용하여 하이퍼파라미터 적절성 평가, 최적값 계산

MNIST의 경우: 훈련 데이터의 20% 검증데이터로 분리 사용

```
(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:]
```

6.5.2 하이퍼파라미터 최적화

•0단계
하이퍼파라미터 값의 범위를 설정
예: 0.001~1000 사이, 10의 거듭제곱 단위로 범위 지정

•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.

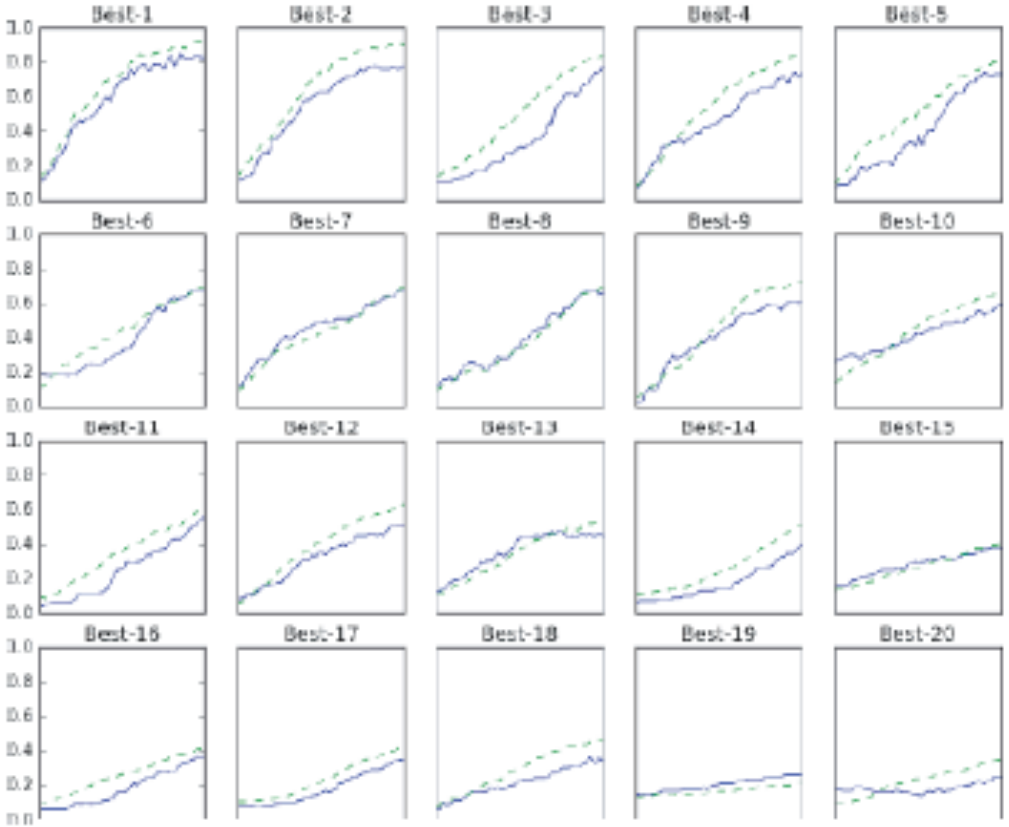
6.5.3 하이퍼파라미터 최적화 구현하기

Weight decay, learning rate:
로그 스케일 무작위 추출

```
weight_decay = 10 ** np.random.uniform(-8, -4)
lr = 10 ** np.random.uniform(-6, -2)
```

lr : 0.001 ~ 0.1
weight_decay: 1e-8 ~ 1e-6
축소된 범위에서 반복

그림 6-24 실선은 검증 데이터에 대한 정확도, 점선은 훈련 데이터에 대한 정확도




```
Best-1 (val acc:0.83) | lr:0.0092, weight decay:3.86e-07
Best-2 (val acc:0.78) | lr:0.00956, weight decay:6.04e-07
Best-3 (val acc:0.77) | lr:0.00571, weight decay:1.27e-06
Best-4 (val acc:0.74) | lr:0.00626, weight decay:1.43e-05
Best-5 (val acc:0.73) | lr:0.0052, weight decay:8.97e-06
```