밑바닥부터 시작하는 딥러닝 CHATER 6

학습 관련 기술들

매개변수 갱신

신경망 학습의 목적은 손실 함수의 값을 낮추는 매개변수를 찾는것이다. 매개변수의 최적값을 찾는다는 것을 최적화라고 한다. 최적의 매개변수를 찾는 단서로 미분을 이용했다. 매개변수의 기울기를 구하고 기울어진 방향으로 매개변수 값을 갱신하는 일을 반복하여 최적의 값을 찾는 것을 확률적 경사 하강법이라고 한다.

확률적 경사 하강법(SGD)

SGD를 코드로 구현해보면 다음과 같다.

```
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]
```

lr은 학습률을 뜻한다. update(params, grads) 메서드는 SGD 과정에서 반복해서 불린다. 인수인 params와 grads는 딕셔너리 변수이다. params['W1'], grads['W1'] 등과 같이 각각 가중치 매개변수와 기울기를 저장하고 있다.

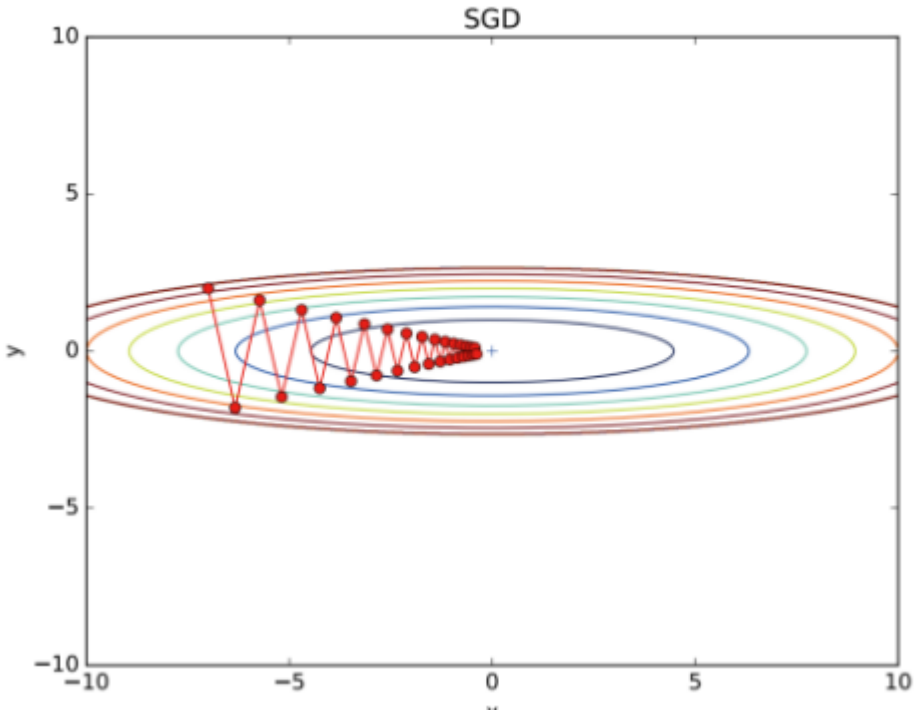
SGD의 단점

SGD는 단순하고 구현하기 쉽지만 문제에 따라서 비효율적일 때가 있다. 예를 들어 다음 함수의 최소값을 구하는 문제를 생각해보자.

$$f(x,y) = \frac{1}{20}x^2 + y^2$$

이것의 최적화 갱신 경로를 구해보면 다음 그림과 같이 그려진다.

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



SGD의 단점은 비등방성(방향에 따라 성질(기울기)가 달라지는 함수)에서는 탐색 경로가 비효율적이라는 것이다. 그렇기 때문에 이를 개선해주는 모멘텀, AdaGrad, Adam 이라는 방법들을 사용한다.

모멘텀

모멘텀은 운동량을 뜻하고 수식으로는 다음과 같이 나타낼 수 있다.

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

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

SGD에서처럼 갱신할 가중치 매개변수, 손실함수, 학습률 등이 있따. v라는 변수는 속도를 뜻한다.

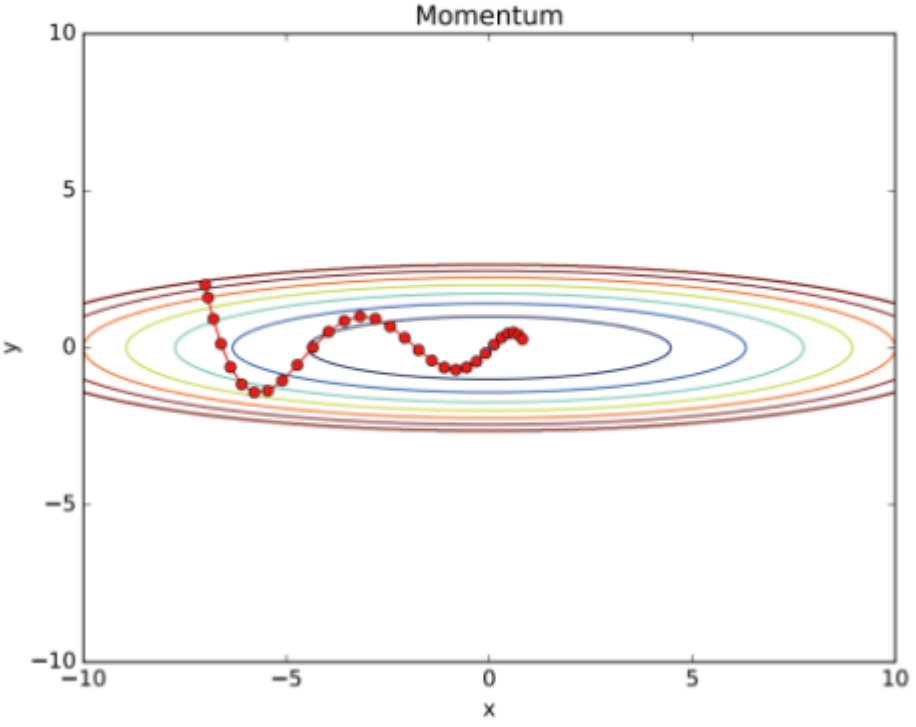
모멘텀을 소스코드로 구현하면 다음과 같다.

```
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]
```

v는 초기화 때는 아무 값도 담지 않고 대신 update()가 처음 호출될 때 매개변수와 같은 구조의 데이터를 딕셔너리 변수로 저장한다. 모멘텀을 통한 최적화 갱신 경로는 다음과 같아진다.

그림 6-5 모멘텀에 의한 최적화 갱신 경로



AdaGrad

신경망 학습에서는 학습률이 중요하다. 이 값이 너무 작으면 학습 시간이 너무 길어지고, 반대로 너무 크면 발산하여 학습이 제대로 이뤄지지 않는다. 이 학습률을 정하는 효과적 기술로 학습률 감소가 있다. 학습을 진행하면서 학습률을점차 줄여가는 방법인데 처음에는 크게 학습하다가 조금씩 작게 학습하는 방법이다. AdaGrad는 각각의 매개변수에 적응적으로 학습률을 조정하면서 학습을 진행한다. 수식으로 나타내면 다음과 같다.

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

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

h는 기존 기울기 값을 제곱하여 계속 더해준다. 그리고 매개변수를 갱신할 때 1/sqrt(h)를 곱하여 학습률을 조정한다. AdaGrad는 과거의 기울기를 제곱하여 계속 더해나가기 때문에 학습을 진행할 수록 갱신 강도가 약해진다. 그래서 무한대로 학습을 해나가면 어느 순간 갱신량이 0이 되어 전혀 갱신이 되지 않느다. 이 문제를 개선한 기법으로 RMSProp 기법이 있다. 이것은 과거의 모든 기울기를 균일하게 더해가는 것이 아니라, 먼 과거의 기울기는 서서히 잊고 새로운 기울기 정보를 크게 반영한다. 지수이동평균이라고도 하며 과거 기울기의 반영 규모를 기하급수적으로 감소시킨다.

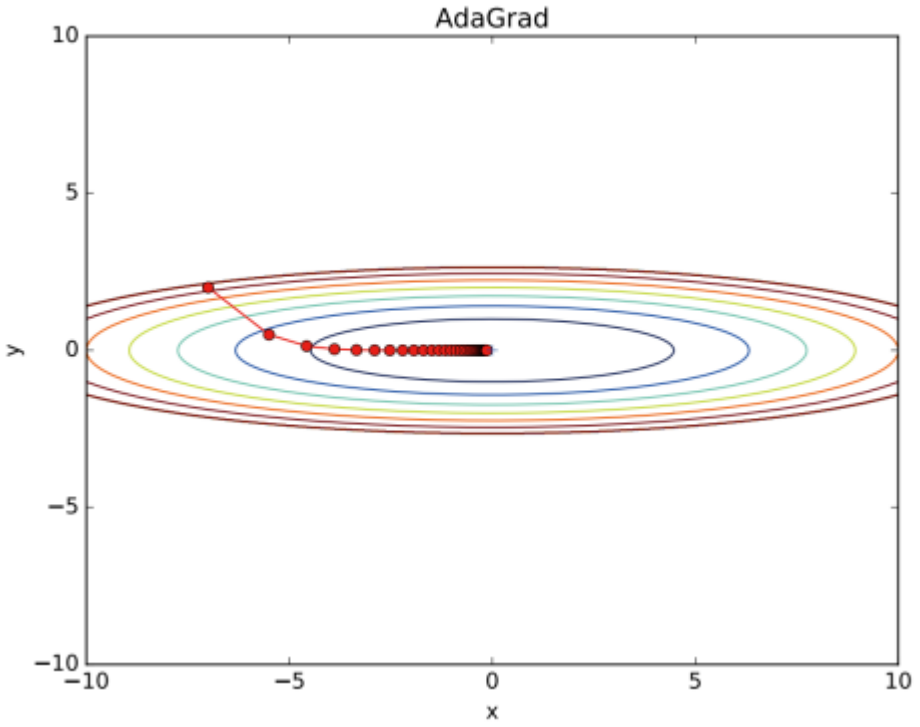
AdaGrad를 소스코드로 구현하면 다음과 같다.

```
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)
```

마지막에 1e-y이라는 작은 값을 더해서 0으로 나누는 상황을 막는다는 것이 중요한 포인트이다. AdaGrad에 의한 최적화 갱신 경로는 다음과 같아진다.

그림 6-6 AdaGrad에 의한 최적화 갱신 경로



Adam

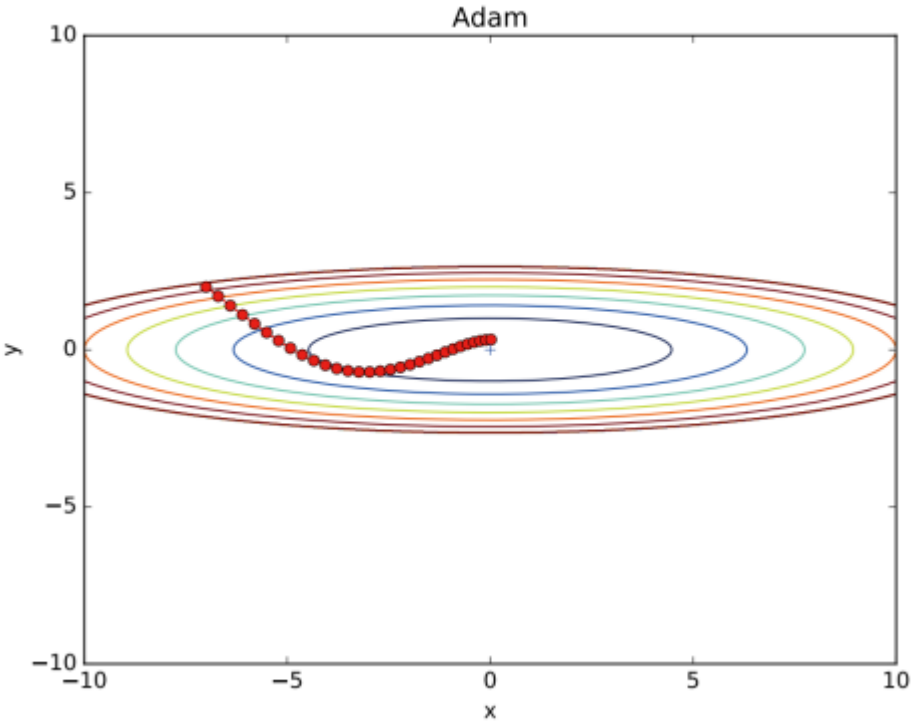
위의 두 기법을 융합한 기법이 Adam 이다. 매개변수 공간을 효율적으로 탐색하면서 하이퍼파라미터의 편향 보정이 진행된다. Adam을 코드로 구현하면 다음과 같다.

```
class Adam:

    """Adam (http://arxiv.org/abs/1412.6980v8)"""

    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.it
er) / (1.0 - self.beta1**self.iter)

        for key in params.keys():
            #self.m[key] = self.beta1*self.m[key] + (1-sel
f.beta1)*grads[key]
            #self.v[key] = self.beta2*self.v[key] + (1-sel
f.beta2)*(grads[key]**2)
            self.m[key] += (1 - self.beta1) * (grads[key] -
self.m[key])
            self.v[key] += (1 - self.beta2) * (grads[key]**
2 - self.v[key])

            params[key] -= lr_t * self.m[key] / (np.sqrt(se
lf.v[key]) + 1e-7)

            #unbias_m += (1 - self.beta1) * (grads[key] - s
elf.m[key]) # correct bias
            #unbisa_b += (1 - self.beta2) * (grads[key]*gra
ds[key] - self.v[key]) # correct bias
            #params[key] += self.lr * unbias_m / (np.sqrt(u
nbisa_b) + 1e-7)
```

Adam에 의한 최적화 갱신 경로는 다음과 같아진다.

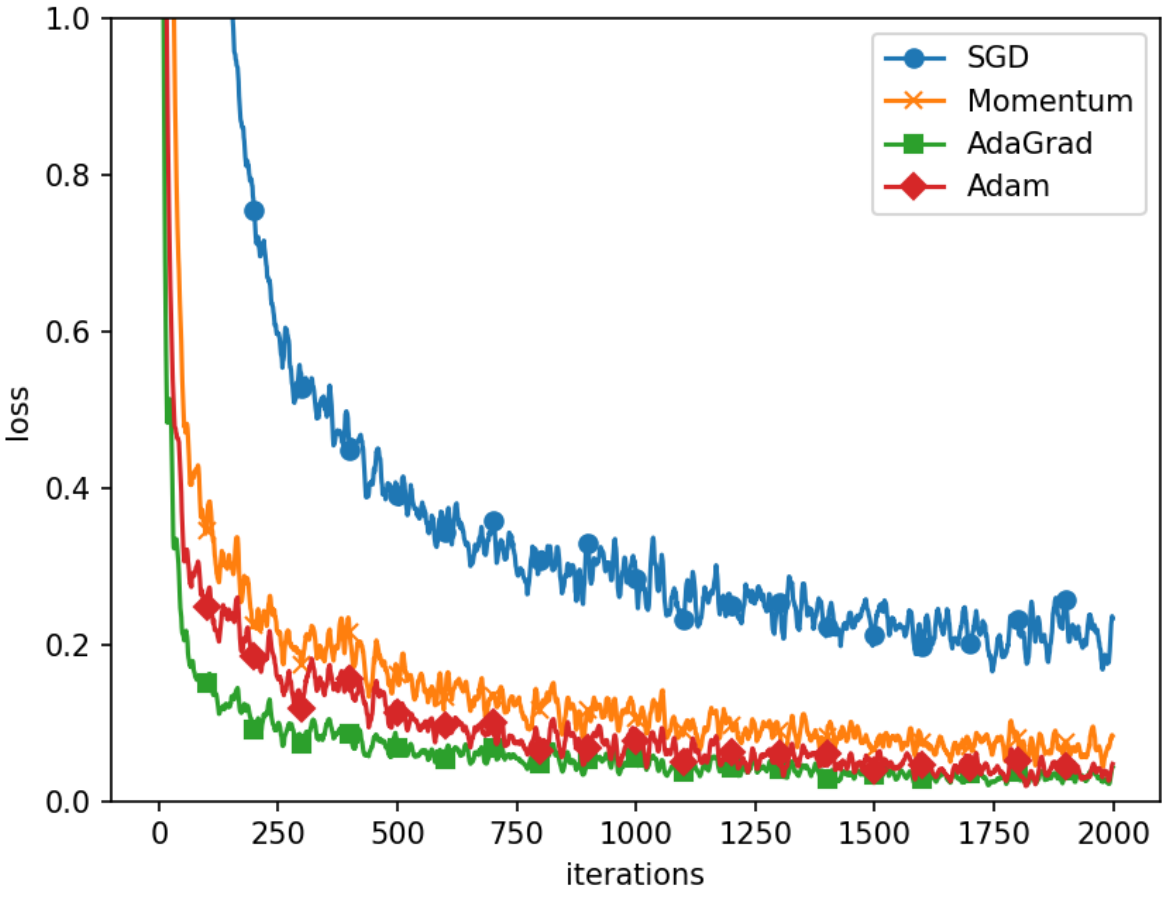
그림 6-7 Adam에 의한 최적화 갱신 경로



Adam은 하이퍼 파라미터를 3개를 설정한다. 지금까지의 학습률, 일차 모멘트용 계수, 이차 모멘트용 계수가 그것들이다.

MNIST 데이터 셋으로 본 갱신 방법 비교

위의 4 방법을 MNIST 데이터 셋을 통해 비교해보면 그래프결과는 다음과 같이 나오게 된다.



하이퍼파라미터인 학습률과 신경망의 구조에 따라 결과가 달라지지만 일반적으로 SGD보다 다른 기법들이 빠르게 학습하고 최종 정확도도 높다.

가중치의 초기값

초기값이 0일 때

overfitting을 억제하여 범용 성능을 높이는 방법으로 가중치 감소 기법이 있다. 가중치 값을 작게 하여 overfitting이 일어나지 않게 하는 것이다. 그러나 초기값이 0이면 오차역전파법에서 모든 가중치의 값이 똑같이 갱신되기 때문에 올바른 학습이 이루어지지 않는다. 그렇기 때문에 초기값을 무작위로 설정해야 한다.

은닉층의 활성화값 분포

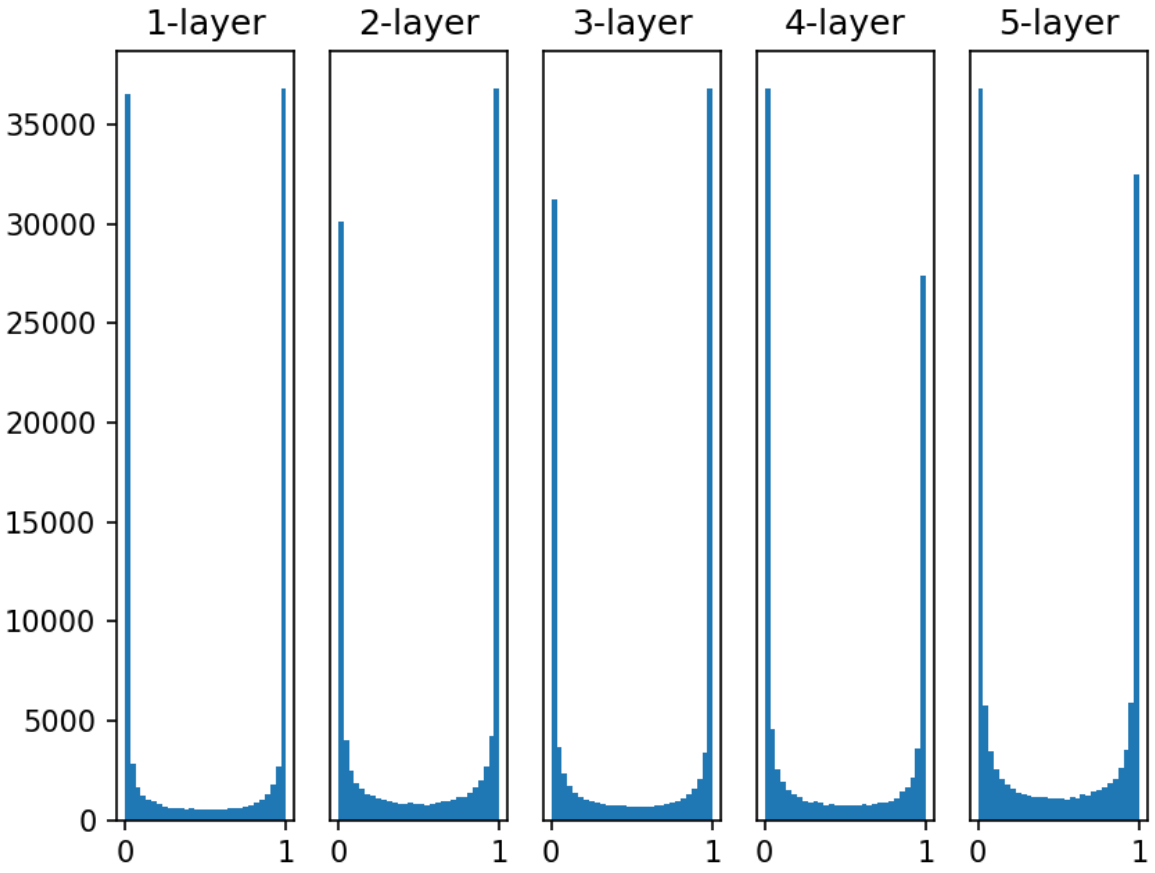
활성화 함수로 시그모이드 함수를 사용하는 5층 신경망에 무작위로 생성한 입력 데이터를 흘리며 각 층의 활성화값 분포를 히스토그램으로 그려 보려고한다.

우선 코드는 다음과 같다.

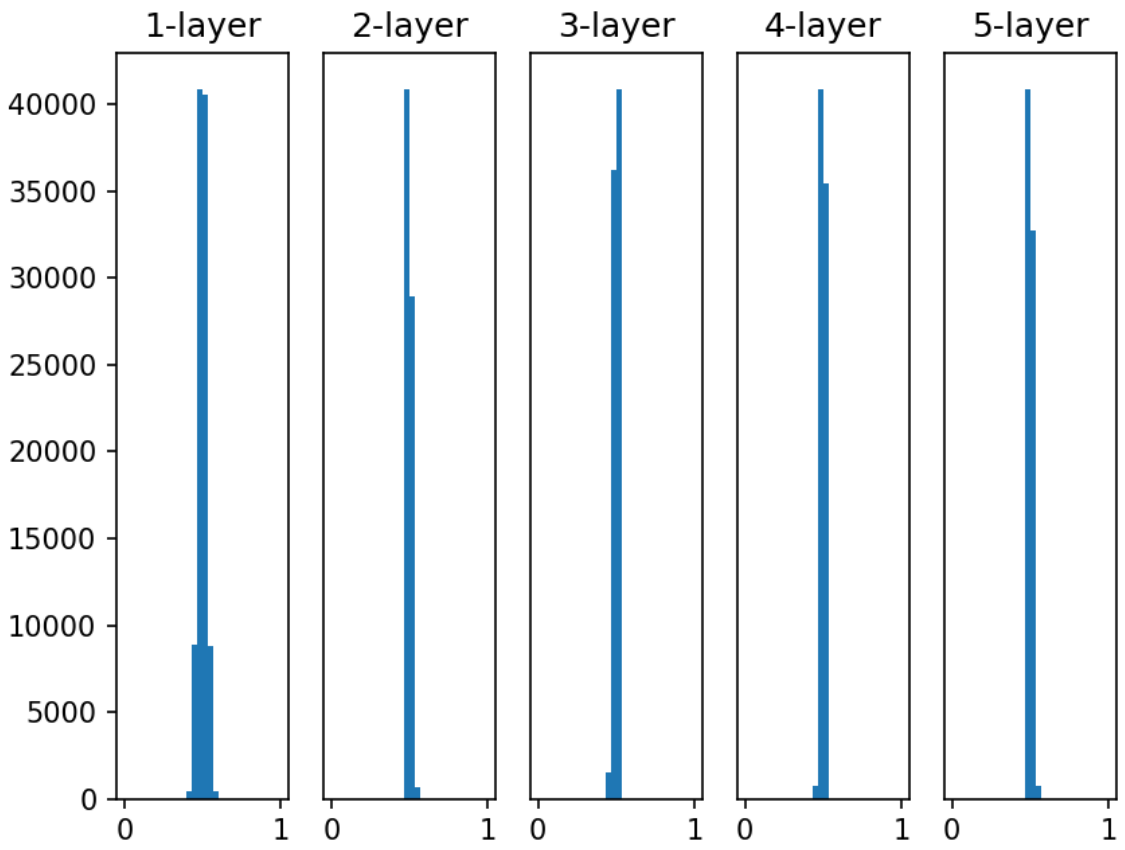
```
import numpy as np
import matplotlib.pyplot as plt


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()
```

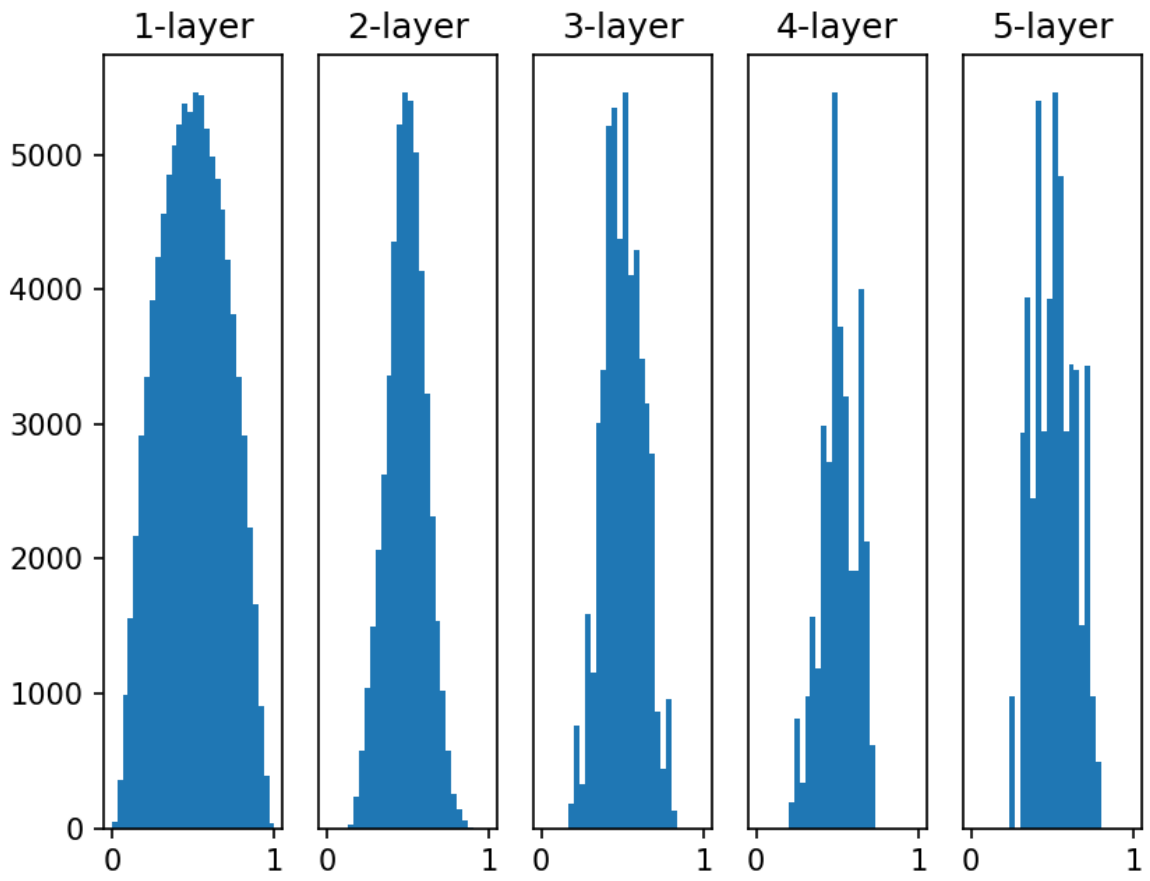
각 층의 활성화 결과를 activations 변수에 저장했다. 그리고 이것을 실행시켜 얻은 결과값은 다음과 같다.



시그모이드 함수는 출력이 0 또는 1에 가까워지면 미분값은 0에 가까워진다. 그래서 데이터가 0과 1에 치우쳐 분포하게 되면 역전파 기울기 값이 점점 작아지다가 사라진다. 이것이 기울기 소실이라고 알려진 문제이다. 만약 가중치의 표준편차를 0.01로 바꿔서 실험하면 다음과 같다.



활성화 값들이 치우쳤다는 것은 다수의 뉴런이 거의 같은 값을 출력하고 있다는 뜻이다. 그렇기 때문에 표현력을 제한한다는 문제가 발생한다. 이어서 Xavier 초기값(표준편차가 1/sqrt(n))을 사용하여 코드를 변경하고 그 값을 출력하면 결과값은 다음과 같다.

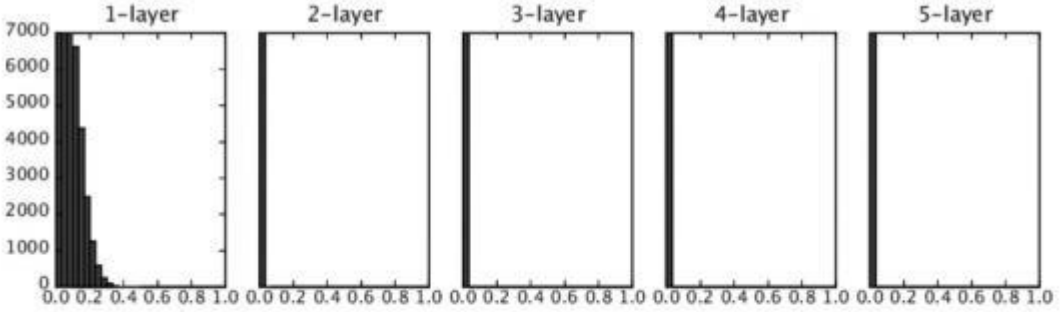


각 층에 흐르는 데이터도 적당히 퍼져 있고 시그모이드 함수의 표현력도 제한받고 있지 않기 때문에 효율적인 학습이 이뤄질 것으로 예상이 된다.

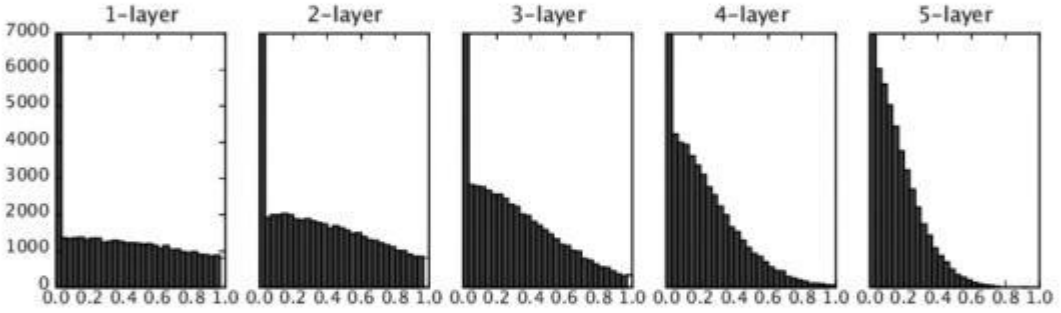
ReLU를 사용할 때의 가중치 초기값

이 초기값은 He 초기값이라고도 하고 앞 계층의 노드가 n개일 때 표준편차가 sqrt(2/n)인 정규분포를 사용한다. 활성화 함수로 ReLU를 이용한 경우의 활성화값 분포를 살펴보면 다음과 같다.

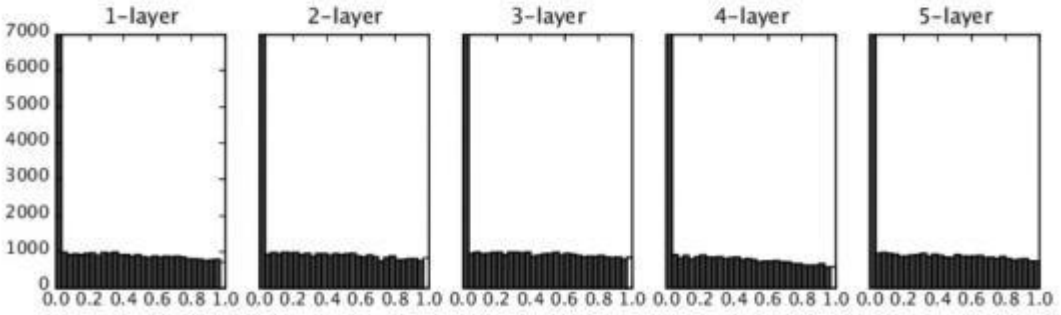
그림 6-14 활성화 함수로 ReLU를 사용한 경우의 가중치 초깃값에 따른 활성화값 분포 변화



표준편차가 0.01인 정규분포를 가중치 초깃값으로 사용한 경우



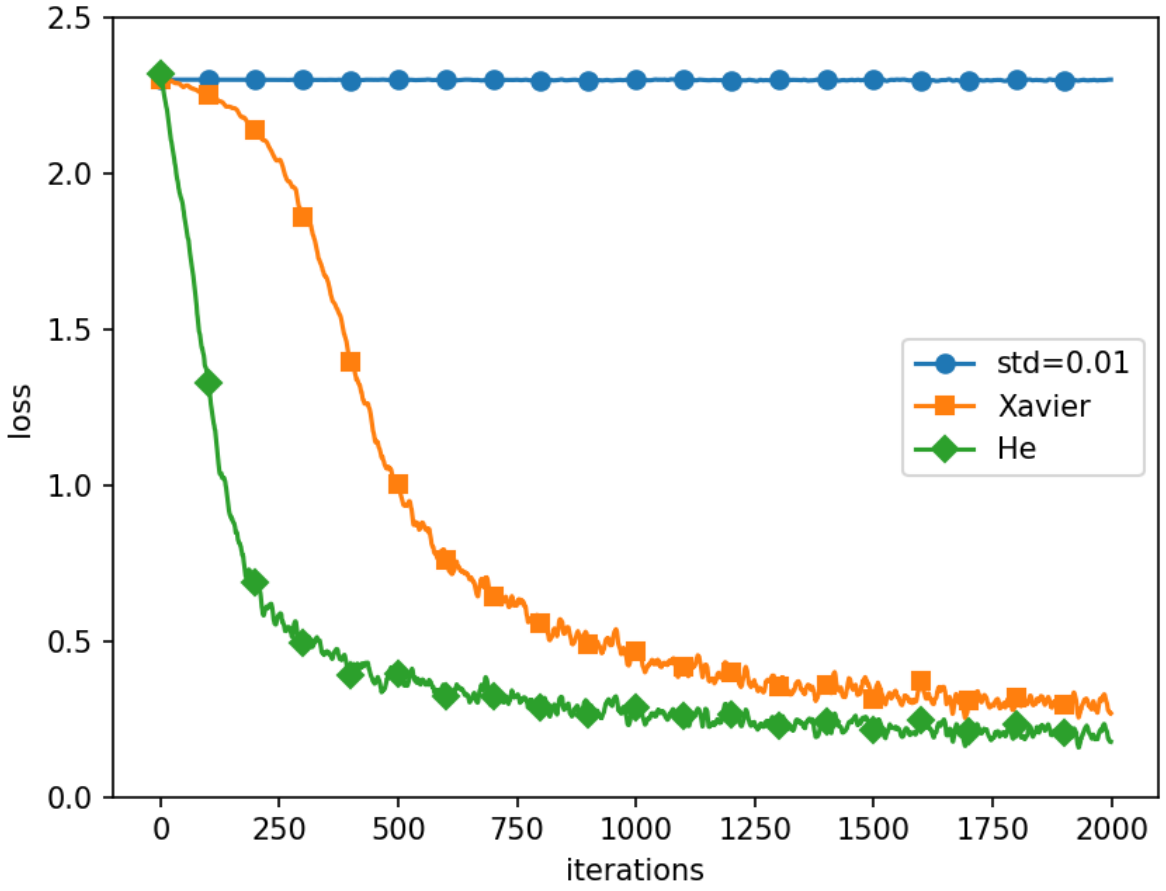
Xavier 초깃값을 사용한 경우



He 초깃값을 사용한 경우

결론적으로 He 초기값을 사용하면서 sigmoid나 tanh 등의 S자 모양 곡선일 때는 Xavier 초기값을 사용하는 것이 바람직하다.

MNIST 데이터셋으로 본 가중치 초기값 비교



std = 0.01 일 때는 순전파 때 너무 작은 값이 흐르기 때문에 학습이 전혀 이뤄지지 않는다. 반대로 Xavier와 He 초기값의 경우 학습이 순조롭게 이뤄진다.

배치 정규화

각 층이 활성화를 적당히 퍼뜨리도록 강제하는 것을 배치 정규화라고 한다.

배치 정규화 알고리즘

배치 정규화의 장점은 크게 3가지로 생각해볼 수 있다.

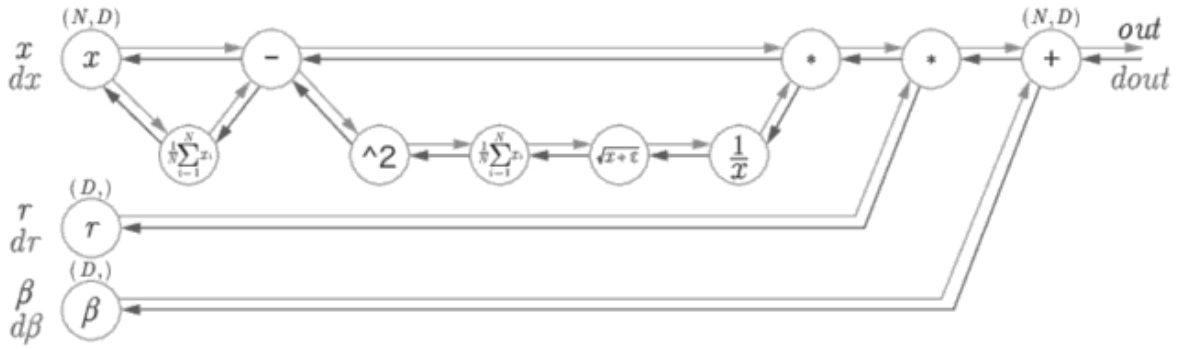
1. 학습 속도 개선

2. 초기값에 크게 의존하지 않음

3. overfitting을 억제할 수 있다.

배치 정규화는 학습 시 미니배치를 단위로 정규화한다. 구체적으로 보면 데이터 분포가 평균이 0, 분산이 1이 되도록 정규화 한다. 또 배치 정규화 계층마다 이 정규화된 데이터에 고유한 확대와 이동변환을 수행한다. 배치 정규화의 계산 그래프는 다음과 같이 나타낼 수 있다.

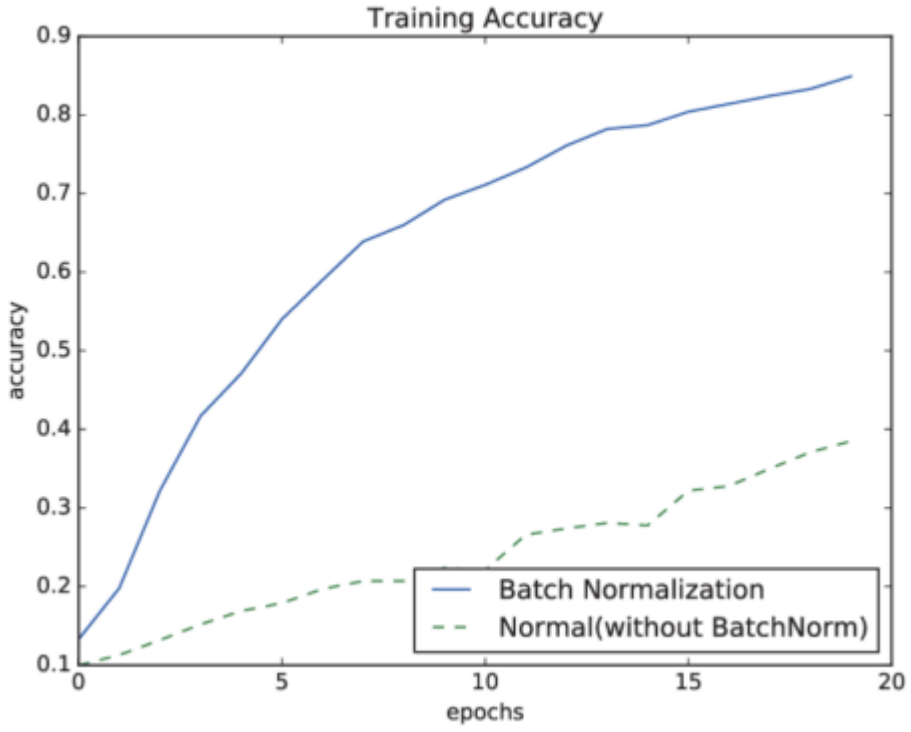
그림 6-17 배치 정규화의 계산 그래프[13]



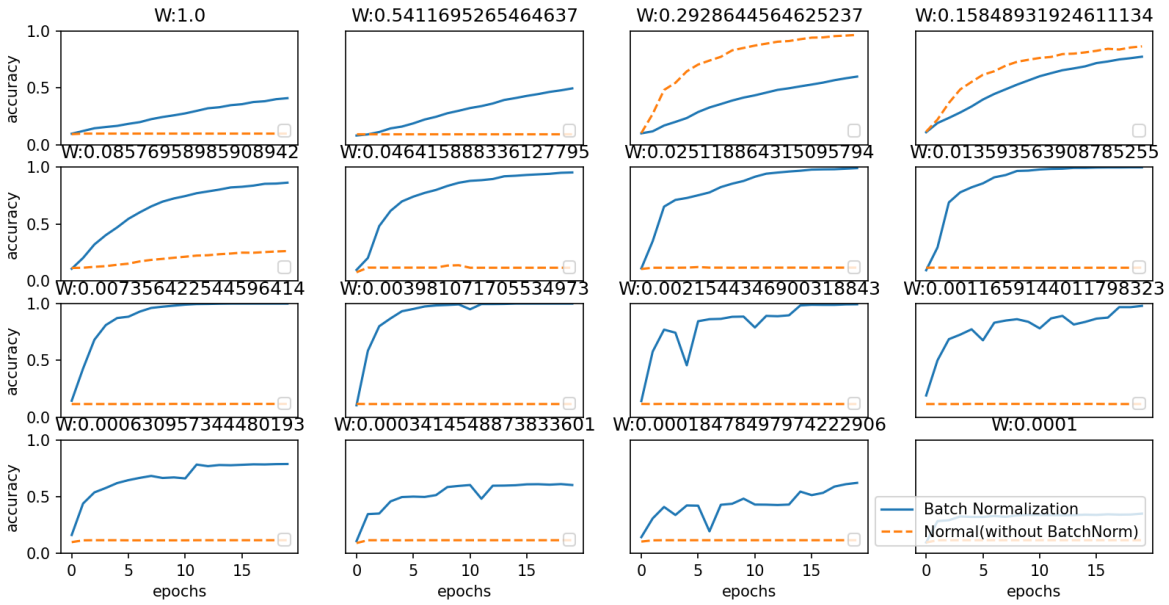
배치 정규화의 효과

MNIST 데이터셋을 활용하여 배치 정규화 계층을 사용할 때와 그렇지 않을 때의 학습 진도 차이를 알아보려 한다.

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



배치 정규화를 사용한 것이 학습을 확연히 빠르게 진전시킨다는 것을 확인할 수 있다. 가중치 초기값 분포를 다양하게 바꿔서 학습을 시키면 결과그래프는 다음과 같이 나오게 된다.



바른 학습을 위해

overfitting이란 신경망이 훈련 데이터에만 지나치게 적응되어 그 외의 데이터에는 제대로 대응하지 못하는 상태를 말한다. 기계학습은 범용 성능을 지향하기 때문에 이를 억제하는 것이 중요하다.

Overfitting

Overfitting은 주로 2가지 경우에 일어난다.

1. 매개변수가 많고 표현력이 높은 모델

2. 훈련 데이터가 적을 경우

코드를 통해 살펴보면 다음과 같다.

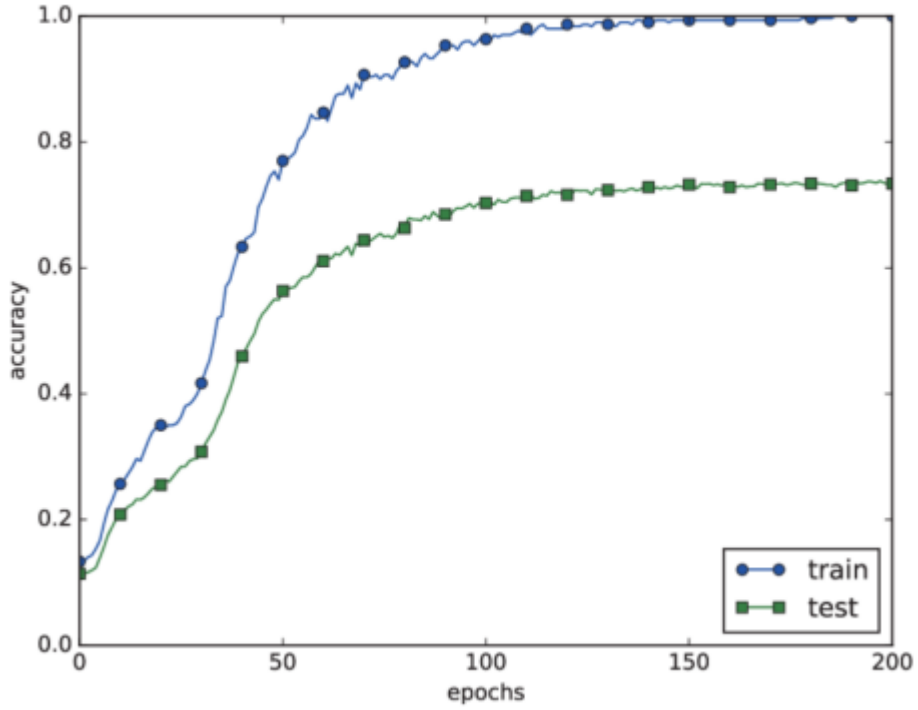
```
import os
import sys

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=[1
00, 100, 100, 100, 100, 100], output_size=10,
                        weight_decay_lambda=weight_decay_la
mbda)
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)

        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', mark
every=10)
plt.plot(x, test_acc_list, marker='s', label='test', markev
ery=10)
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```

train_acc_list와 test_acc_list에는 epoch 단위(모든 훈련 데이터를 한 번씩 본 단위)의 정확도를 저장한다. 위 코드를 실행하면 결과는 다음과 같이 나오게 된다.

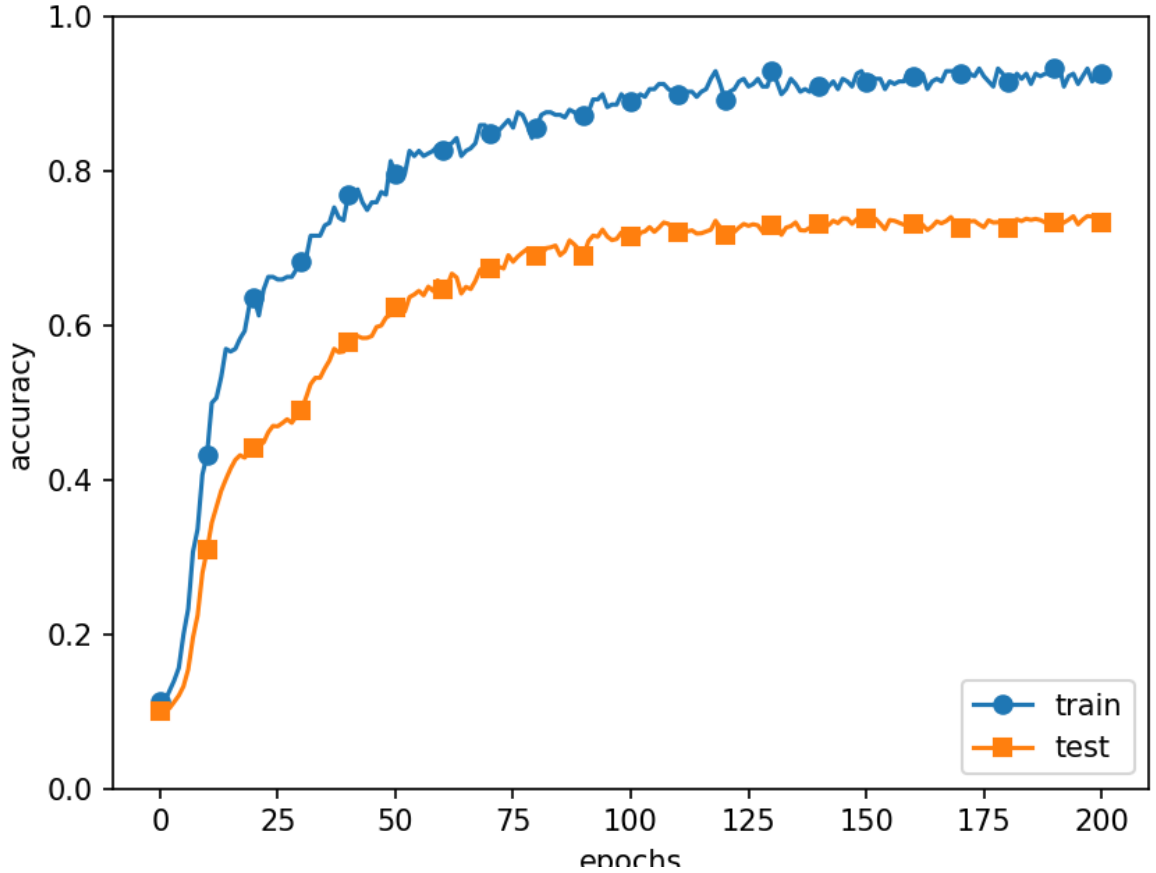
그림 6-20 훈련 데이터(train)와 시험 데이터(test)의 에폭별 정확도 추이



훈련데이터와 시험데이터의 차이가 크게 벌어지게 되는데 이것은 훈련 데이터에만 적응해서 이다.

가중치 감소

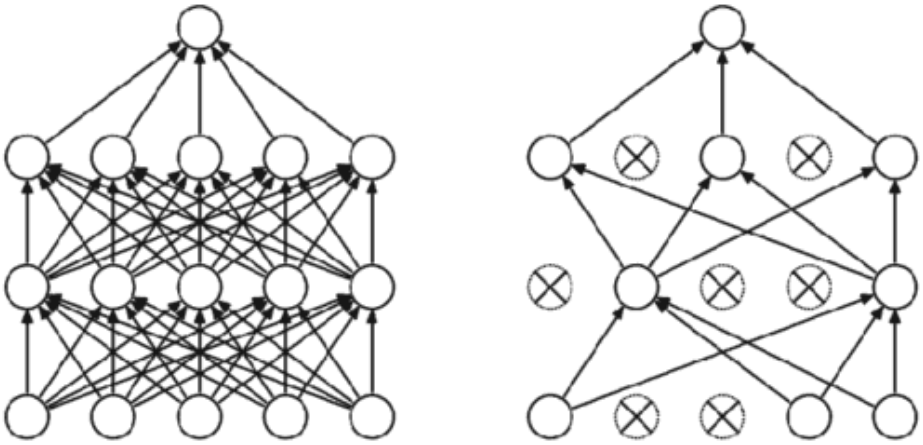
overfitting문제를 억제하기 위해 가중치 감소라는 방법을 사용한다. 학습 과정에서 큰 가중치에 대해서는 큰 페널티를 부과하여 overfitting을 억제하는 방법이다. 가중치 감소는 모든 가중치 각각의 손실 함수에 1/2λW를 더한다 따라서 가중치의 기울기를 구하는 계산에서는 그 동안의 오차역전파법에 따른 정규화 항을 미분한 λW를 더한다. 가중치 감소를 이용하여 훈련 데이터와 시험데이터의 정확도를 비교하면 다음과 같다.



드롭 아웃

신경망 모델이 복잡해지면 가중치 감소만으로는 대응하기 어렵기 때문에 드롭아웃 기법을 사용한다. 드롭아웃은 뉴런을 임의로 삭제하면서 학습하는 방법이다. 훈련 때 은닉층의 뉴런을 무작위로 골라 삭제하고 삭제한 뉴런은 신호를 전달하지 않게 된다. 훈련때는 데이터를 흘릴 때마다 삭제할 뉴런을 무작위로 선택하고 시험 때는 모든 뉴런에 신호를 전달한다. 단 시험 때는 각 뉴런의 출력에 훈련 때 삭제 안 한 비율을 곱하여 출력한다.

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



(a) 일반 신경망 (b) 드롭아웃을 적용한 신경망

드롭 아웃을 코드로 구현하면 다음과 같다.

'''

class Dropout:

"""

http://arxiv.org/abs/1207.0580

"""

def **init**(self, dropout_ratio=0.5):

self.dropout_ratio = dropout_ratio

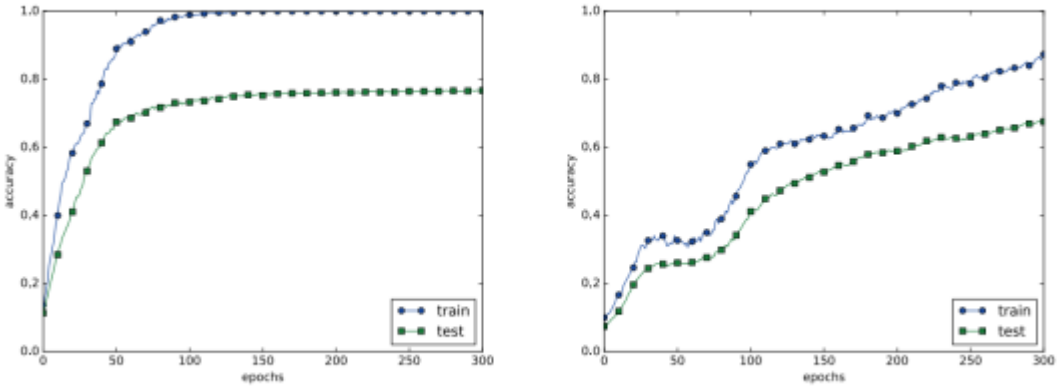
self.mask = None

```
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
```

'''

순전파 때마다 self.mask에 삭제할 뉴런을 False로 표시한다. self.mask는 x와 형상이 같은 배열을 무작위로 생성하고, 그 값이 dropout_ratio보다 큰 원소만 True로 설정한다. 역전파 때의 동작은 ReLU와 같다. 순전파때 신호를 통과시키는 뉴런은 역전파 때도 신호를 그대로 통과시키고 순전파 때 통과시키지 않은 뉴런은 역전파 때도 신호를 차단한다. 이 효과를 MNIST 데이터셋으로 확인하기 위해 코드를 구현하고 실행하면 다음과 같다.

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



드롭아웃을 적용하면 훈련 데이터와 시험 데이터에 대한 정확도 차이도 줄고 훈련 데이터에 대한 정확도가 100%에 도달하지도 않는다. 이처럼 드롭아웃을 이용하면 표현력을 높이면서도 overfitting을 억제할 수 있다.

기계학습에서는 앙상블 학습을 애용한다. 앙상블 학습은 개별적으로 학습시킨 여러 모델의 출력을 평균 내어 추론하는 방식이다. 이 학습방식은 드롭 아웃과 밀접하다.

적절한 하이퍼파라미터 값 찾기

하이퍼파라미터의 예로는 각 층의 뉴런 수, 배치 크기, 매개변수 갱신 시의 학습률과 가중치 감소 등이 있다. 이러한 하이퍼파라미터의 값을 적절히 설정하지 않으면 모델의 성능이 크게 떨어지기도 한다. 그래서 그 값을 적절히 설정하는 것이 매우 중요하다.

검증 데이터

하이퍼파라미터의 성능을 평가할 때에는 시험 데이터를 사용하면 안된다. 그 이유는 시험 데이터를 사용하여 하이퍼파라미터를 조정하면 하이퍼파라미터 값이 시험 데이터에 overfitting되기 때문이다. 그래서 하이퍼파라미터를 조정할 때는 전용 확인 데이터가 필요하다. 그것들을 일반적으로 검증 데이터라고부른다.

훈련데이터는 매개변수 학습

검증 데이터는 하이퍼파라미터 성능 평가

시험 데이터는 신경망의 범용 성능 평가

하이퍼파라미터 최적화

하이퍼파라미터를 최적화할 때의 핵심은 하이퍼파라미터의 최적 값이 존재하는 범위를 조금씩 줄여간다는 것이다. 그것을 위해서 우선 대략적인 범위를 설정하고 그 범위에서 무작위로 하이퍼파라미터 값을 골라낸 후 그 값으로 정확도를 평가한다. 정확도를 잘 살피면서 이 작업을 여러 번 반복하여 하이퍼파라미터의 최적 값의 범위를 좁혀가는 것이다. 단계별로 살펴보면 다음과 같다.

0단계

하이퍼파라미터 값의 범위를 설정한다.

1단계

설정된 범위에서 하이퍼파라미터의 값을 무작위로 추출한다.

2단계

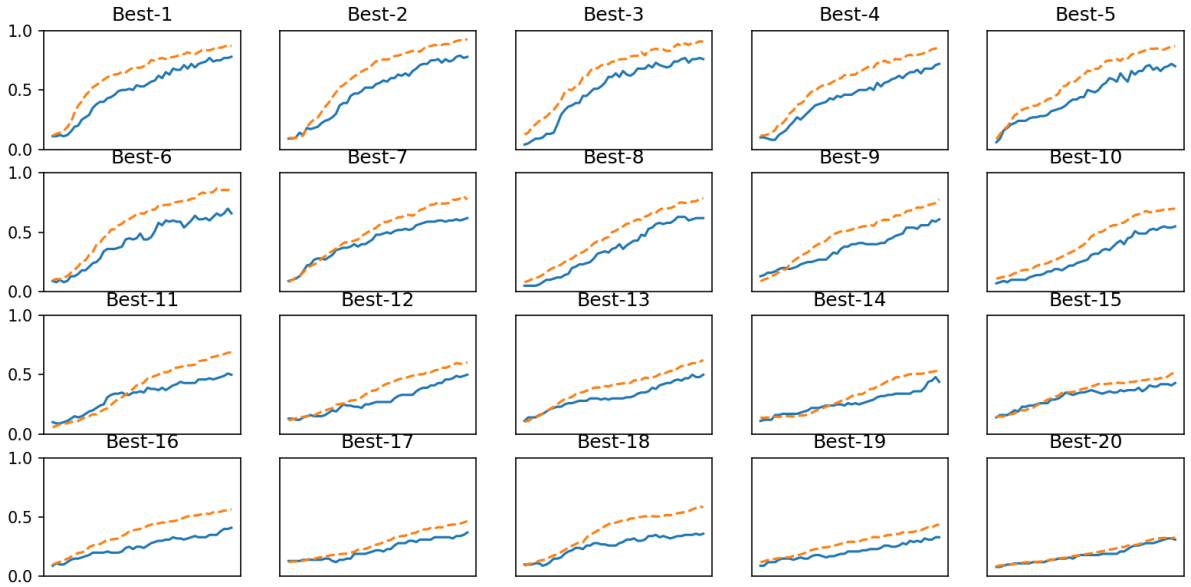
1단계에서 샘플링한 하이퍼파라미터 값을 사용하여 학습하고, 검증 데이터로 정확도를 평가한다.(단, epoch는 작게 설정한다.)

3단계

1단계와 2단계를 특정 횟수 반복하여 그 정확도를 보고 하이퍼파라미터의 범위를 좁힌다.

하이퍼파라미터 최적화 구현하기

MNIST 데이터셋을 사용하여 하이퍼파라미터를 최적화 해보려한다. 하이퍼파라미터의 검증은 그 값을 0.001~1.000 사이 같은 로그 스케일 범위에서 무작위로 추출해 수행한다. 결과 값을 보면 다음과 같다.



이처럼 잘될 것 같은 값의 범위를 관찰하고 범위를 좁혀 나가고 그 축소된 범위로 똑같은 작업을 반복한다. 반복하다가 특정 단계에서 최종 파라미터 값을 하나 선택한다.