

6. 학습관련 기술들

이번 장에서는 신경망 학습의 핵심 개념들에 대해 알아본다. 그 중, 가중치 매개변수의 최적값을 탐색하는 최적화 방법, 가중치 매개변수 초기값, 하이퍼파라미터 설정 방법 등, 모두가 신경망 학습에서 중요한 주제이다. 오버피팅의 대응책인 가중치 감소와 드롭아웃 등의 정규화 방법도 간략히 설명하고 구현해 본다. 마지막으로 최근 많은 연구에서 사용하는 배치 정규화도 짧게 알아본다. 이번 장에서 설명하는 기법을 이용하면 신경망(딥러닝) 학습의 효율과 정확도를 높일 수 있다.

6.1 매개변수 갱신

신경망 학습의 목적은 손실 함수의 값을 가능한 한 낮추는 매개변수를 찾는 것이었다. 이는 곧 매개변수의 최적값을 찾는 문제이며, 이러한 문제를 푸는 것을 **최적화**라고 한다. 안타깝게도 신경망 최적화는 굉장히 어렵다. 매개변수 공간은 매우 넓고 복잡해서 최적의 솔루션을 찾기 어렵다. 수식을 풀어 순식간에 최솟값을 구하는 방법 같은 것은 없다. 게다가 심층 신경망에서는 매개변수의 수가 엄청나게 많아져서 사태는 더욱 심각하다.

우리는 지금까지 최적의 매개변수 값을 찾는 단서로, 매개변수의 기울기(미분)을 이용했다. 매개변수의 기울기를 구해, 기울어진 방향으로 매개변수 값을 갱신하는 일을 몇번이고 반복해서 점점 최적의 값에 다가갔다. 이것이 **확률적 경사 하강법(SGD)**이란 단순한 방법인데(비록 이름은 어렵지만), 매개변수 공간을 무작정 찾는 것 보다 '똑똑한' 방법이다. SGD는 단순하지만, (문제에 따라서는) SGD보다 똑똑한 방법도 있다. 지금부터 SGD의 단점을 알아본 후, SGD와는 다른 최적화 기법을 소개하고자 한다.

6.1.1 모험가 이야기

본론으로 들어가기 전에 이야기를 하나 말해보겠다.

깊은 곳만 찾아가길 좋아하는 모험가가 있다. 게다가 그는 엄격한 '제약 2개로 자신을 돌아했다. 하나는 지도를 보지 않을 것, 눈가리개를 쓰는 것이다. 지도도 없고 보이지도 않으니 어떻게 가겠는가?

최적 매개변수를 탐색하는 우리가 바로 이 모험가와 같은 심정이지 않을까 싶다. 이 상황에서 중요한 단서는 바로 땅의 '기울기'이다. 이에 지금 서 있는 장소에서 가장 크게 기울어진 방향으로 가지는 것이 SGD의 전략이다. 이 일을 반복하면 언젠가 '깊은 곳'에 찾아갈 수 있을지도 모른다.

6.1.2 확률적 경사 하강법(SGD)

최적화 문제의 어려움을 되새기고자 먼저 SGD를 복습해보겠다. 이를 수식으로 표현하면 다음과 같다.

$$W \leftarrow W - \eta \frac{\partial L}{\partial W}$$

여기에서 W 는 갱신할 가중치 매개변수고 $\frac{\partial L}{\partial W}$ 은 W 에 대한 손실 함수의 기울기이다. η 는 학습률을 의미하는데, 실제로는 0.01이나 0.001과 같은 값을 미리 정해서 사용한다. 또 \leftarrow 는 우변의 값으로 좌변의 값을 갱신한다는 의미이다. 위 식에서 보듯 SGD는 기울어진 방향으로 일정 거리만 가겠다는 단순한 방법이다. 그러면 이 SGD를 파이썬 클래스로 구현해보자.

```
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]
```

초기화 때 받는 인수인 lr은 learning rate(학습률)를 뜻한다. 이 학습률을 인스턴스 변수로 유지한다. update(params, grads)는 (지금까지의 신경망 구현과 마찬가지로) 딕셔너리 변수이다. params['W1'], grads['W1'] 등과 같이 각각 가중치 매개변수와 기울기를 저장하고 있다.

SGD 클래스를 사용하면 신경망 매개변수의 진행을 다음과 같이 수행할 수 있다(실제로는 동작하지 않는 코드이다).

```
optimizer = SGD()  
  
for i in range(10000):  
  
    optimizer.update(params, grads)
```

optimizer는 '최적화를 행하는 자'라는 뜻의 단어다. 이 코드에서는 SGD가 그 역할을 한다.

매개변수 갱신은 optimizer가 책임지고, 수행하니 우리는 optimizer에 매개변수와 기울기 정보만 넘겨주면 된다.

이처럼 최적화를 담당하는 클래스를 분리해 구현하면 기능을 모듈화하기 좋다. 예를 들어 소개할 모멘텀이라는 최적화 기법 역시 update(params, grads)라는 공통의 메서드를 갖도록 구현한다. 그때 optimizer = SGD() 문장을 optimizer = Momentum()으로만 변경하면 SGD가 모멘텀으로 바뀌는 것이다.

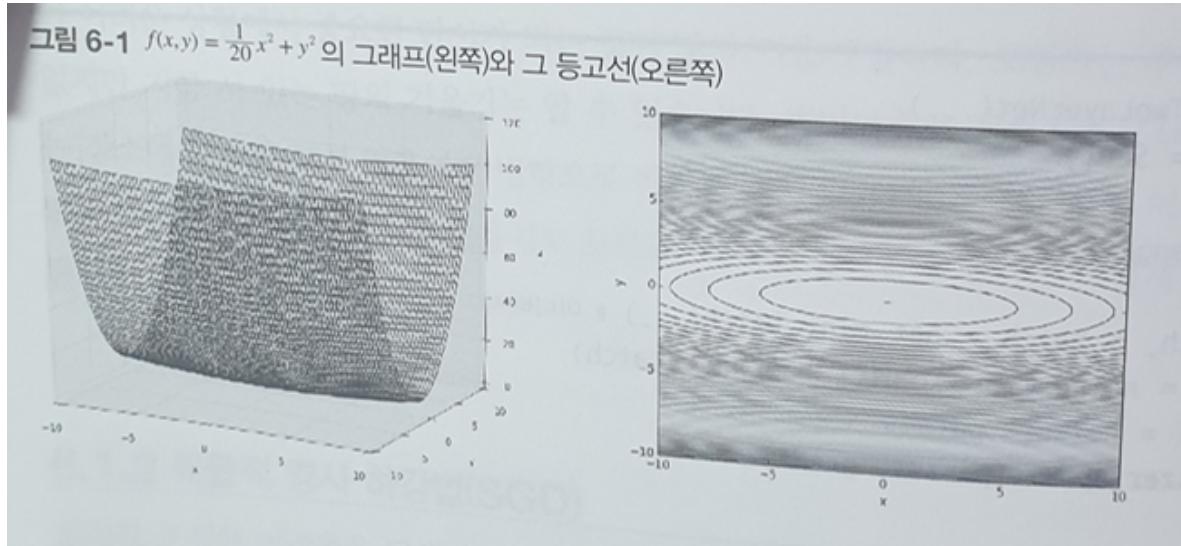
Note 대부분의 딥러닝 프레임워크는 다양한 최적화 기법을 구현해 제공하며, 원하는 기법으로 쉽게 바꿀 수 있는 구조로 되어 있다. 예를 들어, Lasagne라는 딥러닝 프레임워크는 다양한 최적화 기법 구현해 updates.py 파일에 함수로 정리해 놨다. 맘대로 써라.

6.1.3 SGD의 단점

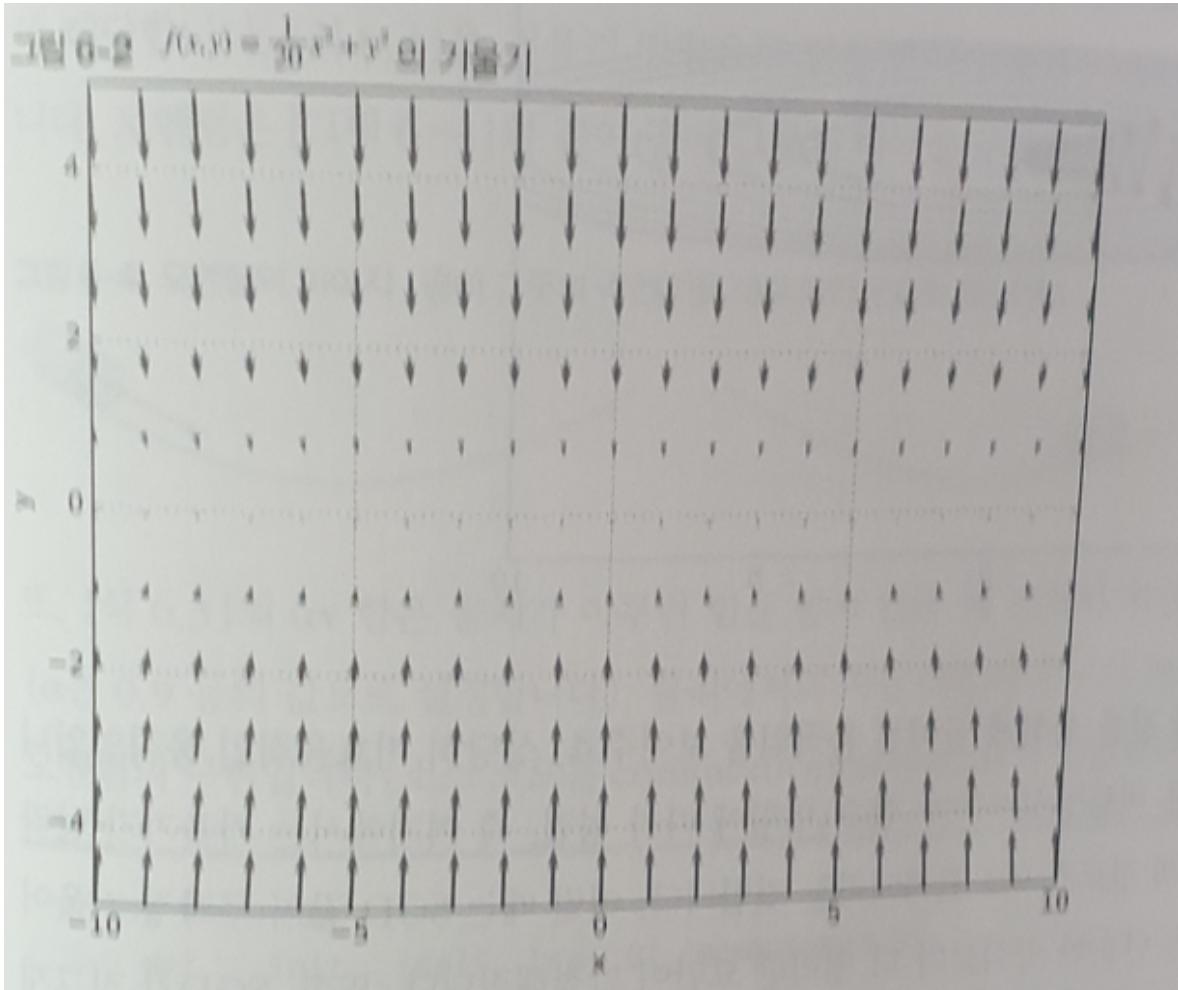
SGD는 단순하고 구현도 쉽지만, 문제에 따라서는 비효율적일 때가 있다. 이번 절에서는 SGD의 단점을 알아보고자 다음 함수의 최솟값을 구하는 문제를 생각해보겠다.

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

이 함수는 아래 그림처럼 '밥 그릇'을 x축 방향으로 늘인 듯한 모습이고, 실제로 그 등고선은 오른쪽과 같이 x축 방향으로 늘인 타원으로 되어 있다.

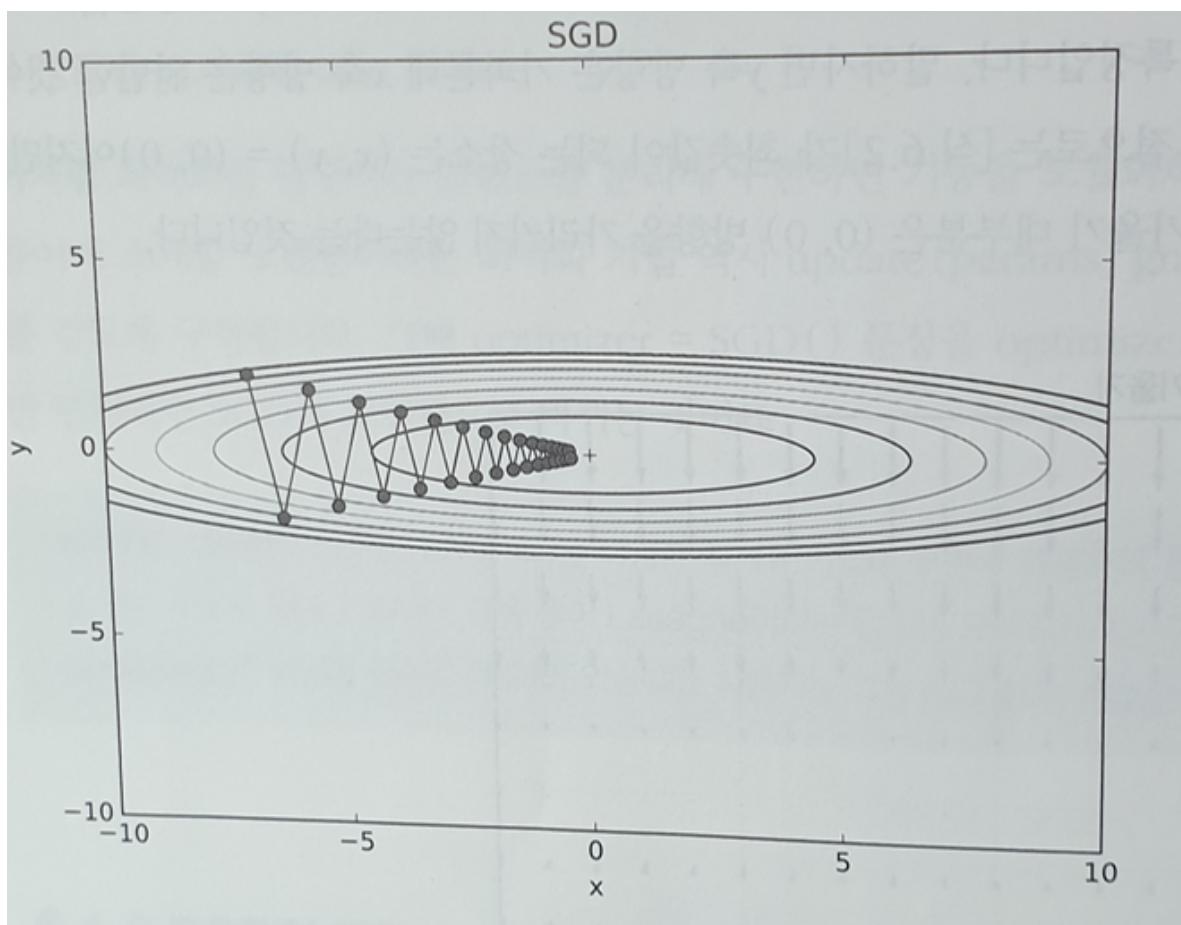


위 함수의 기울기를 그려보면 다음처럼 된다. 이 기울기는 y축 방향은 크고 x축 방향은 작다는 것이 특징이다. 즉, y축 방향은 가파른데 x축 방향은 완만한 것이다. 또, 여기에서 주의할 점은, 최솟값이 되는 장소 (x, y) = (0,0) 이지만, 다음 그림이 보여주는 기울기 대부분은 (0,0) 방향을 가리키지 않는다는 것이다.



이제 함수에 SGD를 적용해보자. 탐색을 시작하는 장소(초깃값)은 $(x,y) = (-7.0, 2.0)$ 으로 하겠다.

결과는 다음처럼 나온다.



SGD는 위와 같은 심하게 굽어진 움직임을 보여준다. 상당히 비효율적이다. 즉, SGD의 단점은 비등방성 함수(방향에 따라 성질, 즉 여기에서는 기울기가 달라지는 함수)에서는 탐색 경로가 비효율적이라는 것이다. 이럴 때는 SGD와 같이 무작정 기울어진 방향으로 진행하는 단순한 방식보다는 더 영리한 묘안이 간절해진다. 또한 SGD가 지그재그로 탐색하는 근본 원인은 기울어진 방향이 본래의 최솟값과 다른 방향을 가리켜서라는 점도 생각해볼 필요가 있다.

이제부터 SGD의 이러한 단점을 개선해주는 모멘텀, AdaGrad, Adam이라는 세 방법을 소개할 것이다. 이들은 모두 SGD를 대체하는 방법으로, 각각을 수식과 파이썬 구현을 통해 알아보자.

6.1.4 모멘텀

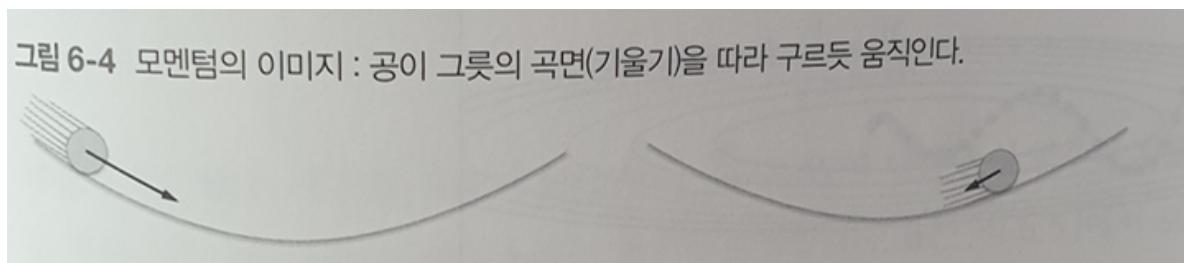
모멘텀은 '운동량'을 뜻하는 단어로, 물리 단어이다. 모멘텀 기법은 수식으로는 다음과 같이 쓸 수 있다.

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

$$W \leftarrow W + v$$

SGD에서처럼 여기서도 W 는 갱신할 가중치 매개변수, 등 다 똑같다.

하지만 여기서는 v 라는 변수가 새로 나오는데, 이는 물리에서 말하는 속도에 해당한다. 이는 기울기 방향으로 힘을 받아 물체가 가속된다는 물리 법칙을 나타낸다. 모멘텀은 공이 그릇의 바닥을 구르는 듯한 움직임을 보여준다.



또, αv 항은 물체가 아무런 힘을 받지 않을 때 서서히 하강시키는 역할을 한다(α 는 0.9 등의 값으로 설정한다). 물리에서의 지면 마찰이나 공기 저항에 해당한다. 다음은 모멘텀 구현이다.

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

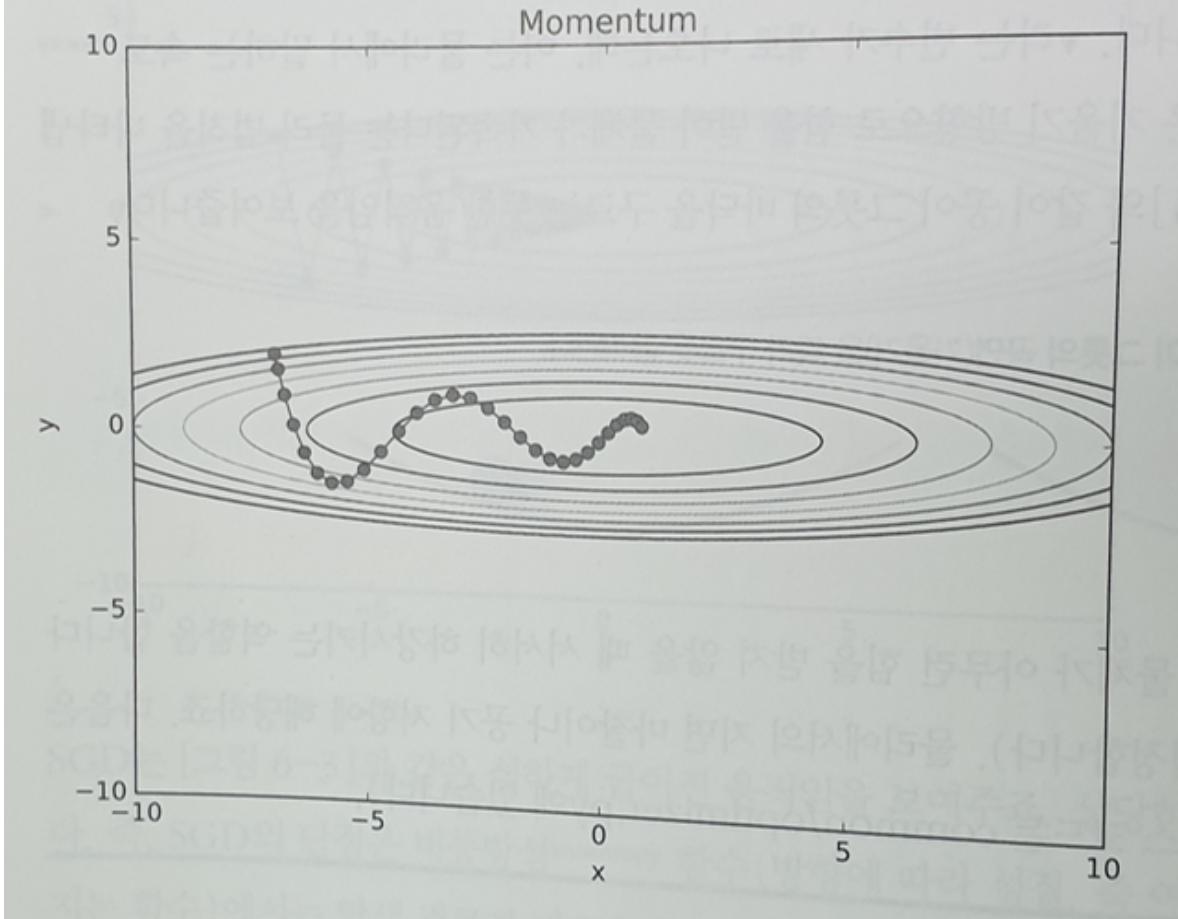
    def update(self, params, grads):
        if self.v is None:
            self.v = {}
        for key, val in params.items():
            self.v[key] = np.zeros_like(val)

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

인스턴스 변수 v 가 물체의 속도이다. v 는 초기화 때는 아무 값도 담지 않고, 대신 `update()`가 처음 호출될 때 매개변수와 같은 구조의 데이터를 딕셔너리 변수로 저장한다. 나머지 부분은 같다.

이제 모멘텀을 사용해서 최적화 문제를 풀어보자.

그림 6-5 모멘텀에 의한 최적화 갱신 경로



그림에서 보듯 모멘텀의 갱신 경로는 공이 그릇 바닥을 구르듯 움직인다. SGD와 비교하면 '지그재그 정도'가 덜한 것을 알 수 있다. 이는 x축의 힘은 아주 작지만 방향은 변하지 않아서 한 방향으로 일정하게 가속하기 때문이다. 거꾸로 y축의 힘은 크지만 위아래로 번갈아 받아서 상충하여 y축 방향의 속도는 안정적이지 않다. 전체적으로는 SGD보다 x축 방향으로 빠르게 다가가 지그재그 움직임이 줄어든다.

6.1.5 AdaGrad

신경망 학습에서는 학습률(수식에서는 η 로 표기) 값이 중요하다. 이 값이 너무 작으면 학습 시간이 너무 길어지고, 반대로 너무 크면 발산하여 제대로 이뤄지지 않는다.

이 학습률을 정하는 효과적 기술로 **학습률 감소**가 있다. 이는 학습을 진행하면서 학습률을 점차 줄여가는 방법이다. 처음에는 크게 학습하다가 조금씩 작게 학습한다는 얘기로, 실제 신경망 학습에 자주 쓰인다. 학습률을 서서히 낮주는 가장 간단한 방법은 매개변수 '전체'의 학습률 값을 일괄적으로 낮추는 것이다. 이를 더욱 발전시킨 것이 AdaGrad이다. AdaGrad는 '각각의' 매개변수에 '맞춤형' 값을 만들어준다. AdaGrad는 개별 매개변수에 적응적으로 학습률을 조정하면서 학습을 진행한다.

AdaGrad의 갱신방법은 수식으로는 다음과 같다.

$$h \leftarrow h + \frac{\partial L}{\partial W} \otimes \frac{\partial L}{\partial W}$$

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

마찬가지로 W 는 갱신할 가중치 매개변수, $\frac{\partial L}{\partial W}$ 은 W 에 대한 손실 함수의 기울기, η 는 학습률을 뜻한다. 여기에서는 새로 h 라는 변수가 등장한다. h 는 첫 번째 식에서도 볼 수 있듯, 기존 기울기 값을 제곱하여 계속 더해준다.

그리고 매개변수를 갱신할 때 $\frac{1}{\sqrt{h}}$ 을 곱해 학습률을 조정한다. 매개변수의 원소 중에서 많이 움직인 (크게 갱신된) 원소는 학습률이 낮아진다는 뜻인데, 다시 말해 학습률 감소가 매개변수의 원소마다 다르게 적용됨을 뜻한다.

Note AdaGrad는 과거의 기울기를 제곱하여 계속 더한다. 이에 학습을 진행할수록 갱신 강도가 약해진다. 실제로 무한히 계속 학습한다면 어느 순간 갱신량이 0이 되어 전혀 갱신되지 않는다. 이 문제를 개선한 기법으로서 RMSProp 이라는 방법이 있다. RMSProp은 과거의 모든 기울기를 균일하게 더해가는 것 아니라, 먼 과거의 기울기는 서서히 잊고 새로운 기울기 정보를 크게 반영한다. 이를 "지수이동평균"이라 하여, 과거 기울기의 반영 규모를 기하급수적으로 감소시킨다.

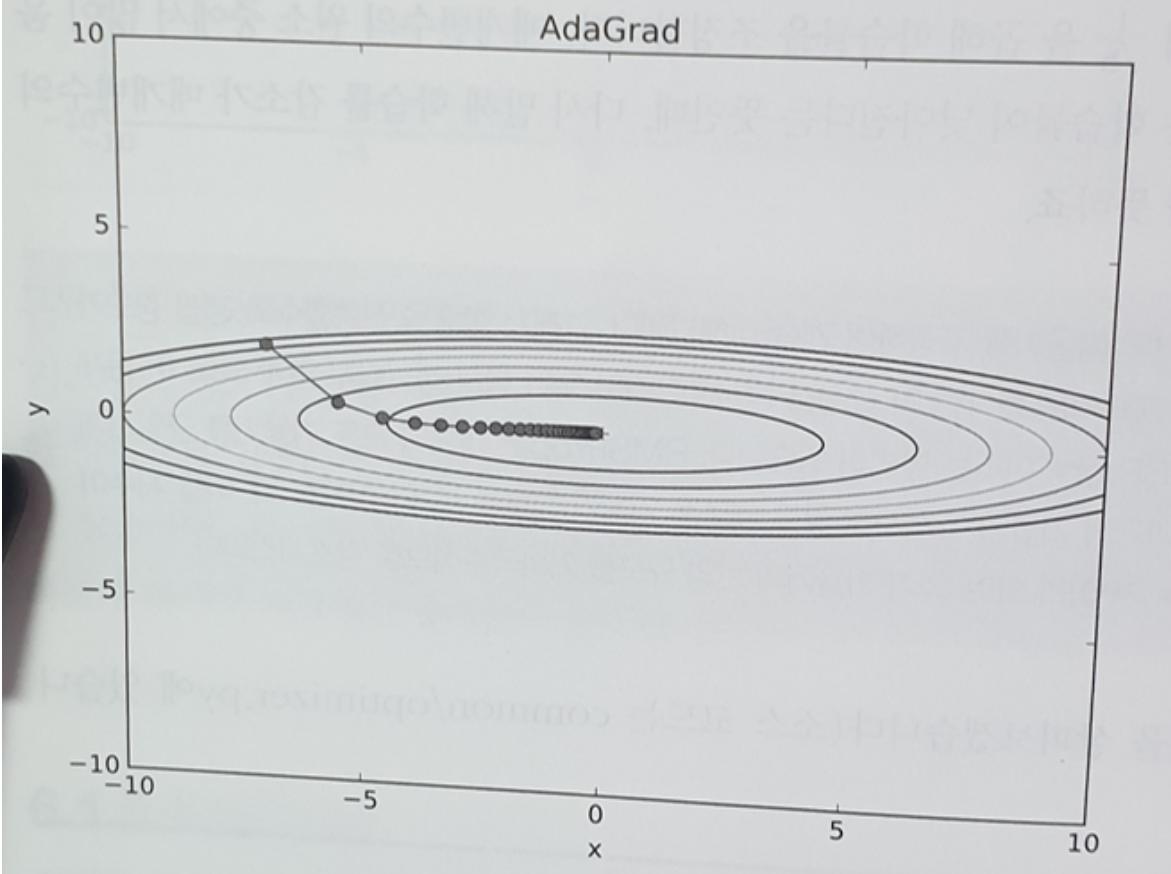
그럼 AdaGrad의 구현을 살펴보자.

```
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] * grad[key]  
            params[key] -= self.lr * grad[key] / (np.sqrt(self.h[key]) + 1e-7)
```

여기에서 주의할 것은 마지막 줄에서 1e-7이라는 작은 값을 더했다는 것이다. 이 작은 값은 설령, self.h[key]에 0이 담겨 있다 해도 0으로 나누는 사태를 막아준다. 대부분의 딥러닝 프레임워크에서는 이 값도 인수로 설정할 수 있다.

그럼 AdaGrad를 사용해서 앞선 최적화 문제를 풀어보겠다.

그림 6-6 AdaGrad에 의한 최적화 갱신 경로

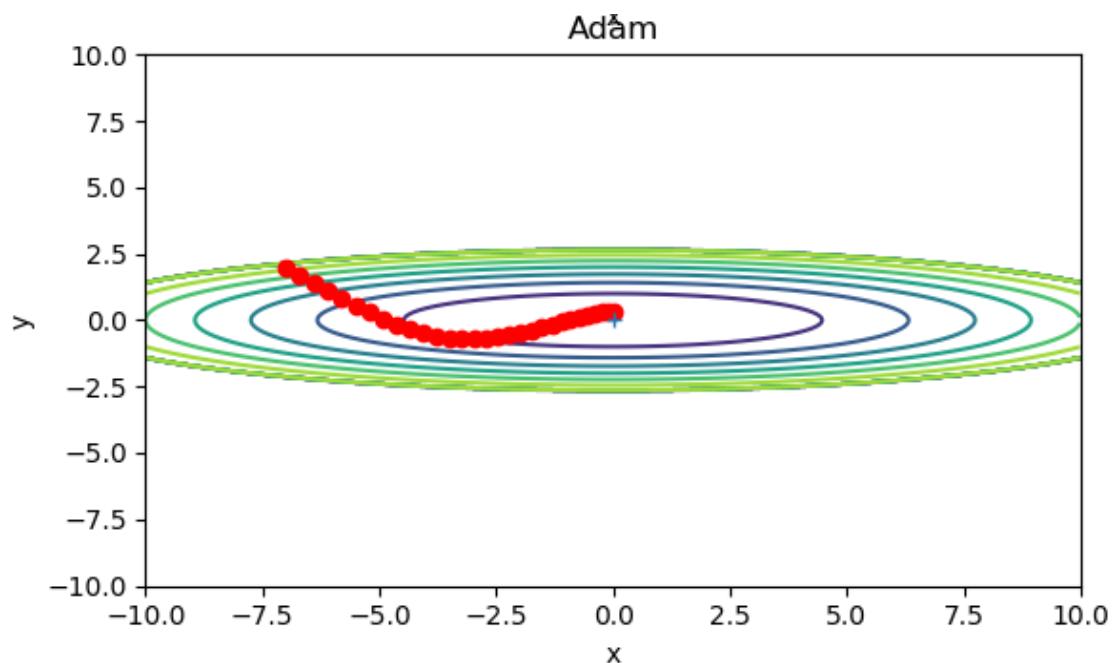


위 그림을 보면 최솟값을 향해 효율적으로 움직이는 것을 알 수 있다. y축 방향은 기울기가 커서 처음에는 크게 움직이지만, 그 큰 움직임에 비례해 갱신 정도도 큰 폭으로 작아지도록 조정된다. 이에 y축 방향으로 갱신 강도가 빠르게 약해지고, 지그재그 움직임이 줄어든다.

6.1.6 Adam

모멘텀은 공이 그릇 바닥을 구르는 듯한 움직임을 보였다. AdaGrad는 매개변수의 원소마다 적응적으로 갱신 정도를 조정했다. Adam은 이 두 기법을 모두 적용한 것이다.

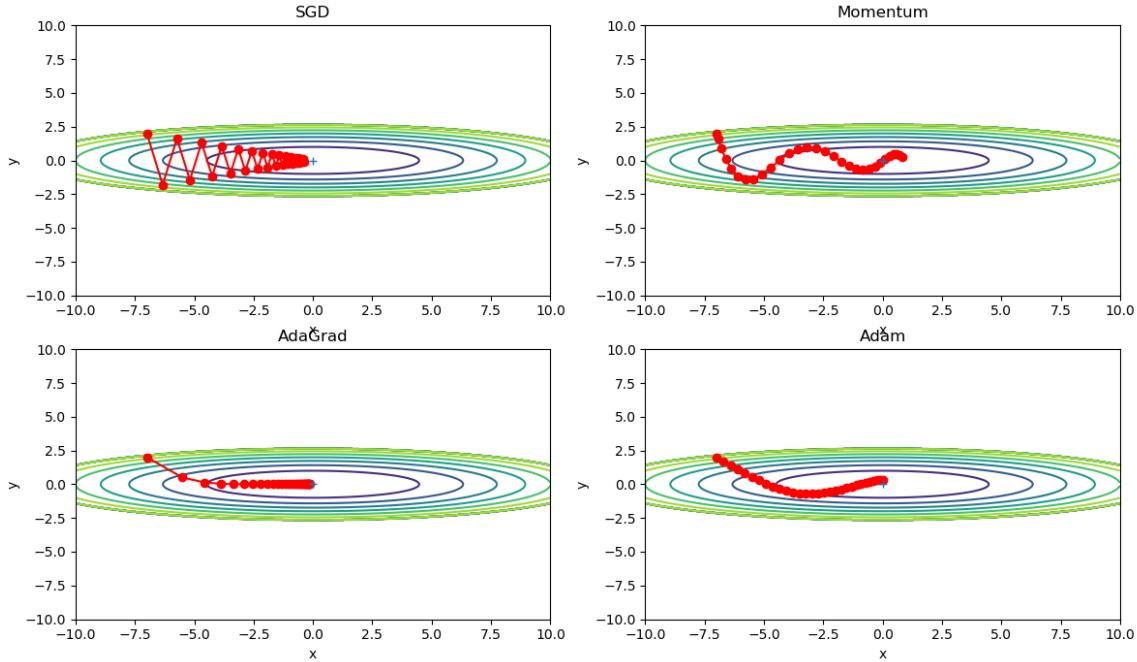
이론이 다소 복잡하다. 하이터 파라미터의 '편향 조정'이 진행된다는 점도 Adam의 특징이다.



Note Adam은 하이퍼파라미터를 3개 설정한다. 하나는 지금까지의 학습률(논문에서는 α 로 등장), 나머지 두 개는 일차 모멘텀용 계수 β_1 , 이차 모멘텀용 계수 β_2 이다. 논문에 따르면 기본 설정값은 $\beta_1 = 0.9, \beta_2 = 0.999$ 이며, 이 값이면 많은 경우에 좋은 결과를 얻을 수 있다.

6.1.7 어느 갱신 방법을 사용할 것인가?

지금까지 매개변수의 갱신 방법을 4개 살펴봤다. 이번 절에서는 이들 네 기법의 결과를 비교해보겠다.



위와 같이 사용한 기법에 따라 갱신 경로가 다르다. 이 그림만 보면 AdaGrad가 가장 나아 보이는데, 사실 그 결과는 풀어야 할 문제가 무엇이냐에 따라 달라진다. 또한 당연하지만 학습률에 따라 결과가 달라진다. 어렵지만 모든 문제에서 항상 뛰어난 기법이라는 것은 없다. 알아서 잘 사용해라.

6.1.8 MNIST 데이터셋으로 본 갱신 방법 비교

손글씨 숫자 인식을 대상으로 지금까지 설명한 기법을 비교해보자.

```
#optimizer_compare_mnist.py

# coding: utf-8
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] = []

# 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]

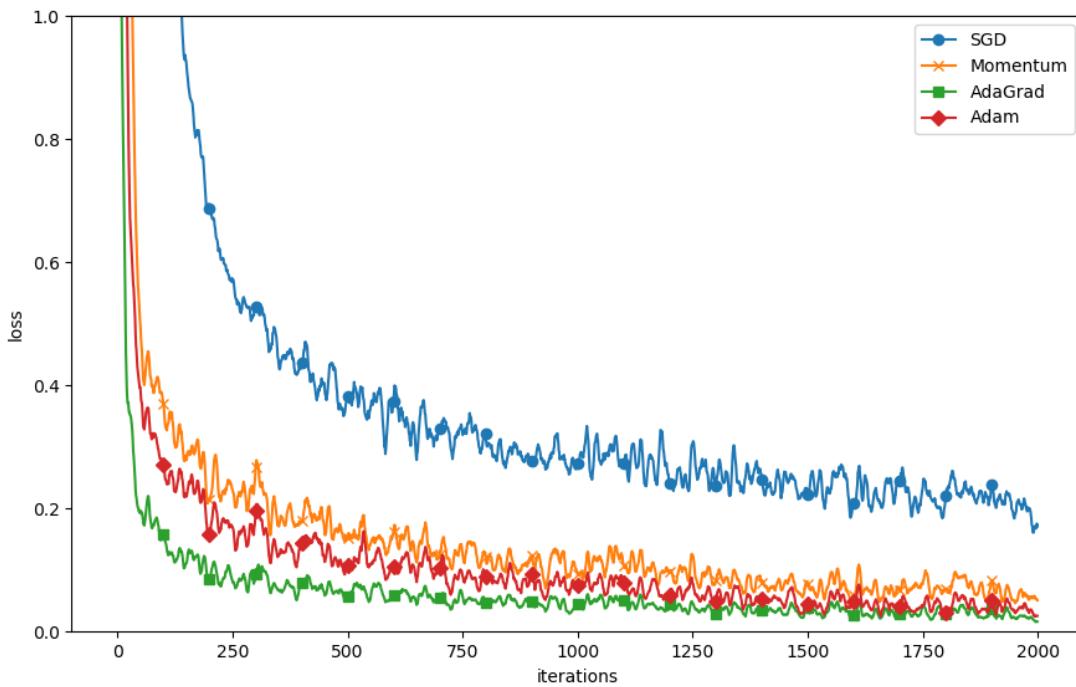
    for key in optimizers.keys():
        grads = networks[key].gradient(x_batch, t_batch)
        optimizers[key].update(networks[key].params, grads)

    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()

```



이 실험은 각 층이 100개의 뉴런으로 구성된 5층 신경망에서 ReLU를 활성화 함수로 사용해 측정했다.

위 결과를 보면 SGD의 학습 진도가 가장 느리다. 나머지 세 기법의 진도는 비슷한데 AdaGrad가 조금 더 빠른 듯 보인다. 이 실험에서 주의할 점은 하이퍼파라미터인 학습률과 신경망의 구조(층 깊이 등)에 따라 결과가 달라진다는 것이다. 다만 일반적으로 SGD보다 다른 세 기법이 빠르게 학습하고, 때로는 최종 정확도도 높게 나타난다.

6.2 가중치의 초기값

신경망 학습에서 특히 중요한 것이 가중치의 초기값이다. 이번 절에서는 권장 초기값에 대해 설명하고, 실험을 통해 실제로 신경망 학습이 신속하게 이뤄지는 모습을 확인하겠다.

6.2.1 초기값을 0으로 하면??

이제부터 오버피팅을 억제해 범용 성능을 높이는 테크닉인 **가중치 감소** 기법을 소개하려 한다. 가중치 감소는 간단히 말하면 가중치 매개변수의 값이 작아지도록 학습하는 방법이다. 가중치 값을 작게하여 오버피팅이 일어나지 않게 하는 것이다.

가중치를 만들고 싶으면 초기값도 최대한 같은 값에서 시작하는 것이 바람직하다. 사실 지금까지 가중치의 초기값은 $0.01 * np.random.randn(10,100)$ 처럼 정규분포에서 생성되는 값을 0.01배 한 작은 값(표준편차가 0.01인 정규분포)을 사용했다.

그렇다면 가중치의 초기값을 0으로 설정하면 어떨까? 매우 명청한 짓이다. 학습이 올바르게 실행되지 않는다. 왜?? 가중치를 균일한 값으로 설정해서는 안된다.

오차역전파법에서 모든 가중치의 값이 똑같이 갱신되기 때문이다. 예를 들어 2층 신경망에서 첫 번째와 두 번째 층의 가중치가 0이라고 가정하겠다. 그럼 순전파 때는 입력층의 가중치가 0이기 때문에 두 번째 층의 뉴런에 모두 같은 값이 전달된다. 두 번째 층의 모든 뉴런에 같은 값이 입력된다는 것은 역전파 때 두 번째 층의 가중치가 모두 똑같이 갱신된다는 말이 된다. 이에 가중치들은 같은 초기값에서 시작하고 갱신을 거쳐도 여전히 같은 값을 유지하는 것이다. 이는 가중치를 여러 개 갖는 의미를 사라지게 한다. 이 '가중치가 고르게 되어버리는 상황'을 막으려면 (정확히는 가중치의 대칭적인 구조를 무너뜨리려면) 초기값을 무작위로 설정해야 한다.

6.2.2 은닉층의 활성화값 분포

은닉층의 활성화값(활성화 함수의 출력 데이터) *의 분포를 관찰하면 중요한 정보를 얻을 수 있다. 이번 절에서는 가중치의 초기값에 따라 은닉층 활성화값들이 어찌 변하는지 간단한 실험을 해보려 한다. 구체적으로는 활성화 함수로 시그모이드 함수를 사용하는 5층 신경망에 무작위로 생성한 입력 데이터를 훌리며 각 층의 활성화값 분포를 히스토그램으로 그려보겠다.

```
# coding: utf-8
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

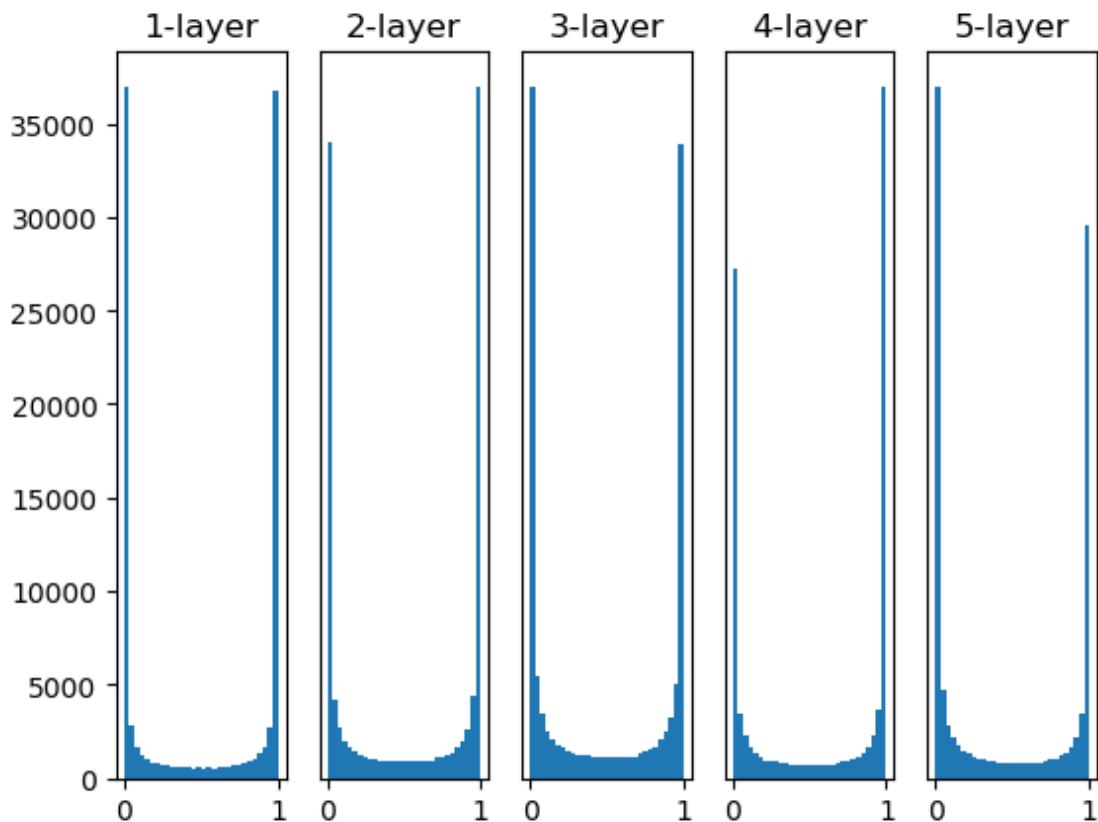
# 히스토그램 그리기
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    if i != 0: plt.yticks([], [])
    # plt.xlim(0.1, 1)
    # plt.ylim(0, 7000)
```

```

plt.hist(a.flatten(), 30, range=(0,1))
plt.show()

```

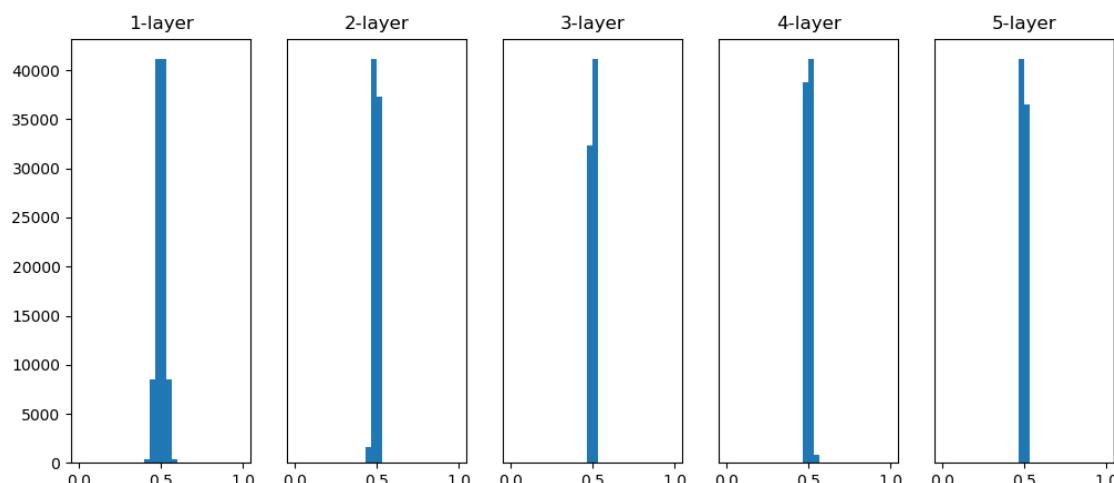
층이 5개가 있으며, 각 층의 뉴런은 100개 쪽이다. 입력 데이터로서 1,000개의 데이터를 정규분포로 무작위로 생성하여 이 5층 신경망에 흘린다. 활성화 함수로는 시그모이드 함수를 이용했고, 각 층의 활성화 결과를 activations 변수에 저장한다. 이 코드에서는 가중치의 분포에 주의해야 한다. 이번에는 표준편차가 1인 정규분포를 이용했는데. 이 분포된 정도(표준편차)를 바꿔가며 활성화값들의 분포가 어떻게 변화하는지 관찰하는 것이 본 실험의 목적이다. 다음은 본 코드의 결과이다.



각 층의 활성화값들이 0과 1에 치우쳐 분포되어 있다. 여기에서 사용한 시그모이드 함수는 그 출력이 0에 가까워지면(또는 1에 가까워지면) 그 미분은 0에 다가간다. 이에 데이터가 0과 1에 치우쳐 분포하면 역전파의 기울기 값이 점점 작아지다가 사라진다. 이것이 바로 **기울기 소실**에 대한 문제이다.

층을 깊게 하는 딥러닝에서 기울기 소실은 더 심각한 문제가 될 수 있다.

이번에는 가중치의 표준편차를 0.01로 바꿔 같은 실험을 반복해보겠다. 앞의 코드에서 가중치 초기값 설정 부분을 다음과 같이 바꾸면 된다.

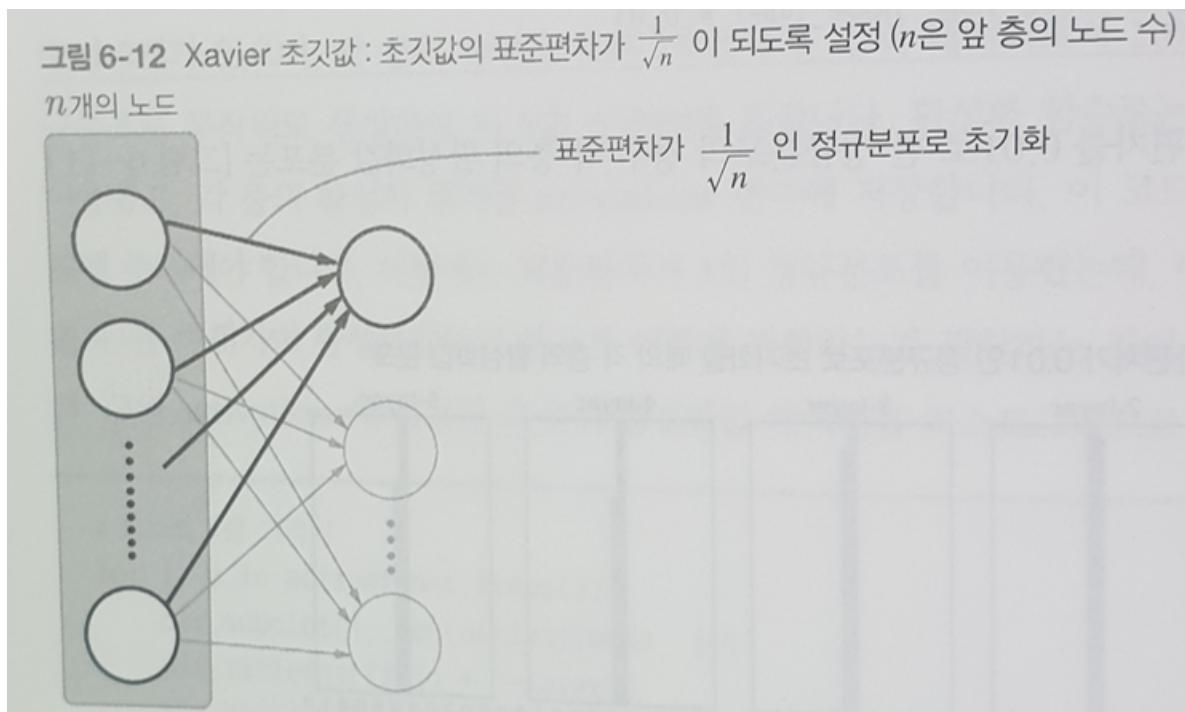


이번에는 0.5 부근에 집중되었다. 앞의 예처럼 0과 1로 치우친 않았으니 기울기 소실 문제는 일어나지 않는다. 활성화값들이 치우쳤다는 것은 표현력 관점에서는 큰 문제가 있다. 이 상황에서는 다수의 뉴런이 거의 같은 값을 출력하고 있으니 뉴런을 여러 개 둔 의미가 없어진다는 뜻이다. 예를 들어 뉴런 100개가 거의 같은 값을 출력한다면 뉴런 1개짜리와 별반 다를 게 없는 것이다. 그래서 활성화값들이 치우치면 표현력을 제한한다는 관점에서 문제가 된다.

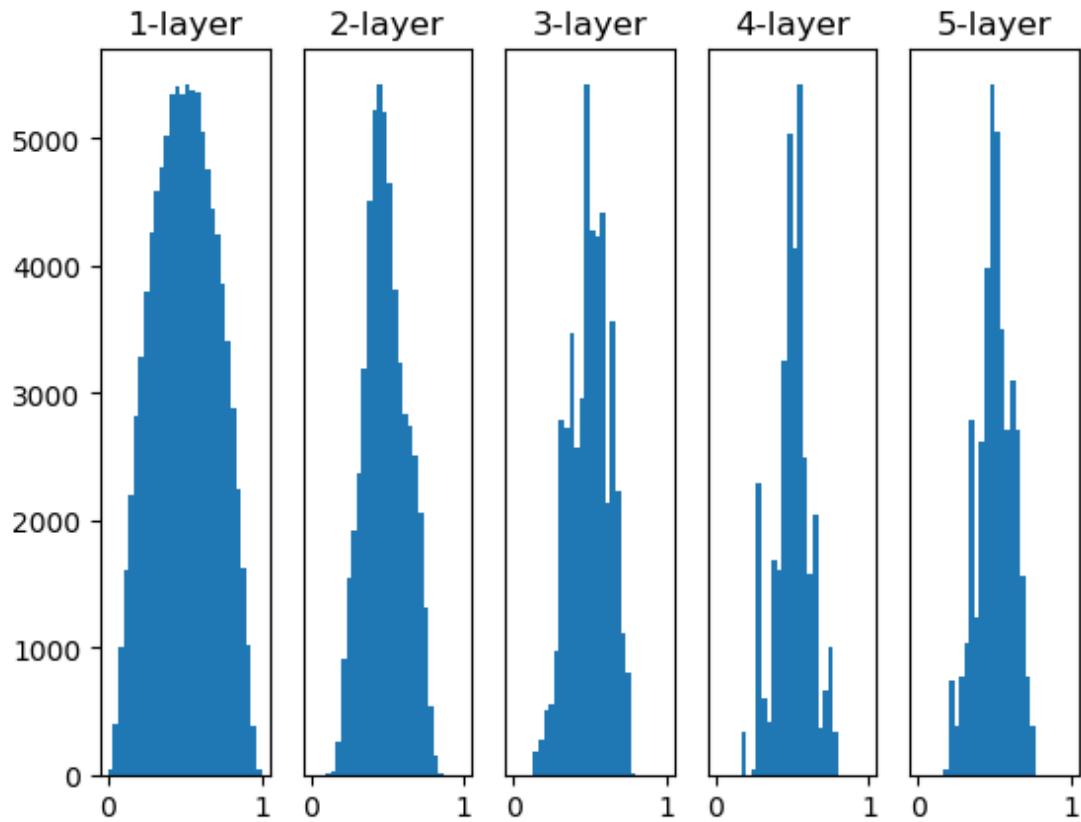
WARNING 각 층의 활성화값은 적당히 고루 분포되어야 한다. 층과 층 사이에 적당하게 다양한 데이터가 흐르게 해야 신경망 학습이 효율적으로 이뤄지기 때문이다. 반대로, 치우친 데이터가 흐르면 기울기 소실이나 표현력 제한 문제에 빠져서 학습이 잘 이뤄지지 않는 경우가 생긴다.

이어 일명 **Xavier 초기값**을 써보겠다. 현재 Xavier 초기값은 일반적인 딥러닝 프레임워크들이 표준적으로 이용하고 있다. 예를 들어 카페 프레임워크는 가중치 초기값을 설정할 때 인수로 xavier 초기값을 지정할 수 있다.

이 논문은 각 층의 활성화값들을 광범위하게 분포시킬 목적으로 가중치의 적절한 분포를 찾고자 했다. 앞 계층의 노드가 n 개라면 표준편차가 $\frac{1}{\sqrt{n}}$ 인 분포를 사용하면 된다는 결론을 이끌었다.



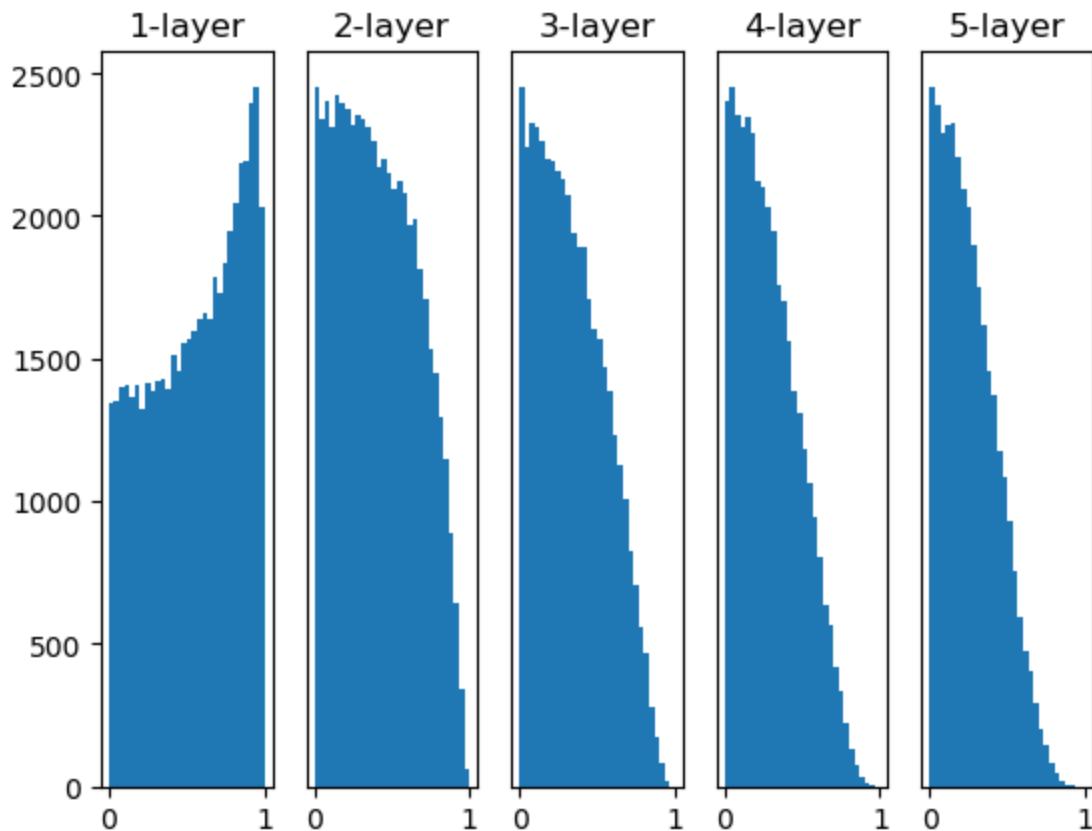
Xavier 초기값을 사용하면 앞 층에 노드가 많을수록 대상 노드의 초기값으로 설정하는 가중치가 좁게 퍼진다. 이제 Xavier 초기값을 써서 실험해보자. 코드에서는 가중치 초기값 설정 부분을 다음과 같이 고쳐주기만 하면 된다(모든 층의 노드 수가 100개라고 단순화했다).



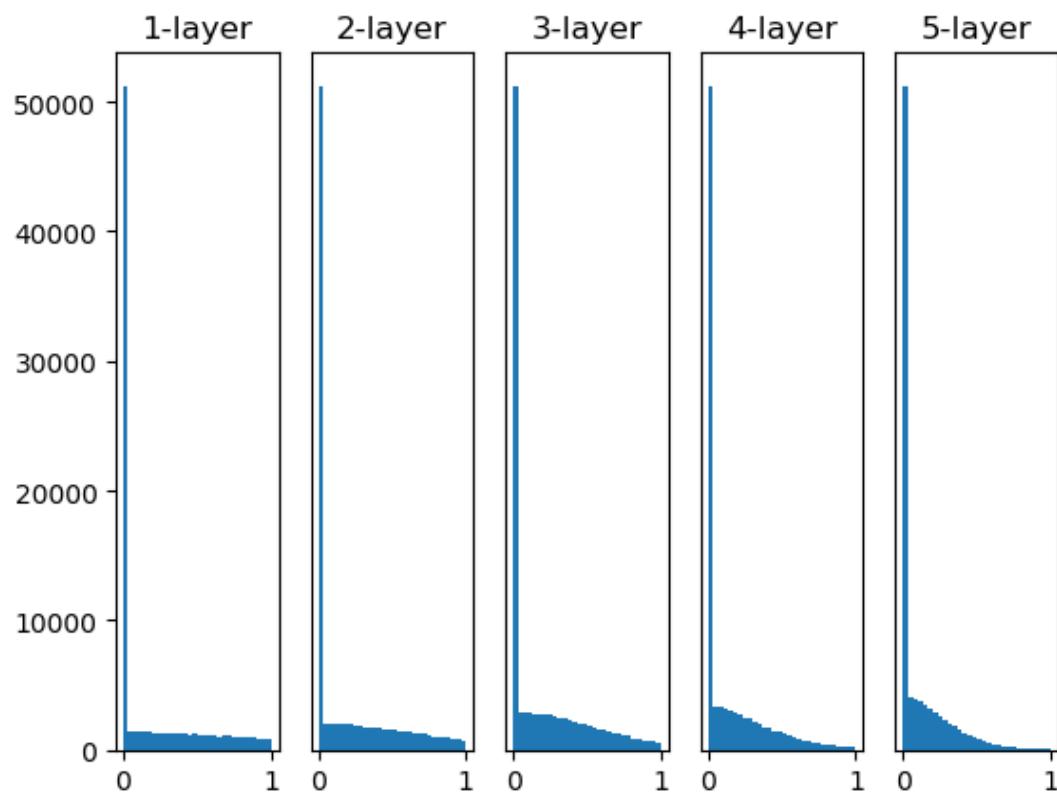
Xavier 초기값을 사용한 결과는 위와 같다. 이 결과를 보면 층이 깊어지면서 형태가 다소 일그러지지만, 앞에서 본 방식보다는 확실히 넓게 분포됨을 알 수 있다. 각 층에 흐르는 데이터는 적당히 퍼져 있으므로 시그모이드 함수의 표현력도 제한받지 않고 학습이 효율적으로 이뤄질 것으로 기대된다.

Note 위 그림은 오른쪽으로 갈수록 약간씩 일그러지고 있다. 이 일그러짐은 sigmoid 함수 대신 tanh 함수(쌍곡선 함수)를 이용하면 개선된다. 실제로 tanh 함수를 이용하면 말끔한 종 모양으로 분포된다. tanh 함수도 sigmoid 함수와 같은 'S' 자 모양 곡선 함수다. 다만 tanh 함수가 원점(0,0)에서 대칭인 S 곡선인 반면, sigmoid 함수는 $(x,y) = (0,0.05)$ 에서 대칭인 S 곡선이다. 활성화 함수용으로는 원점에서 대칭인 함수가 바람직하다고 알려져 있다.

tanh 함수 이용시



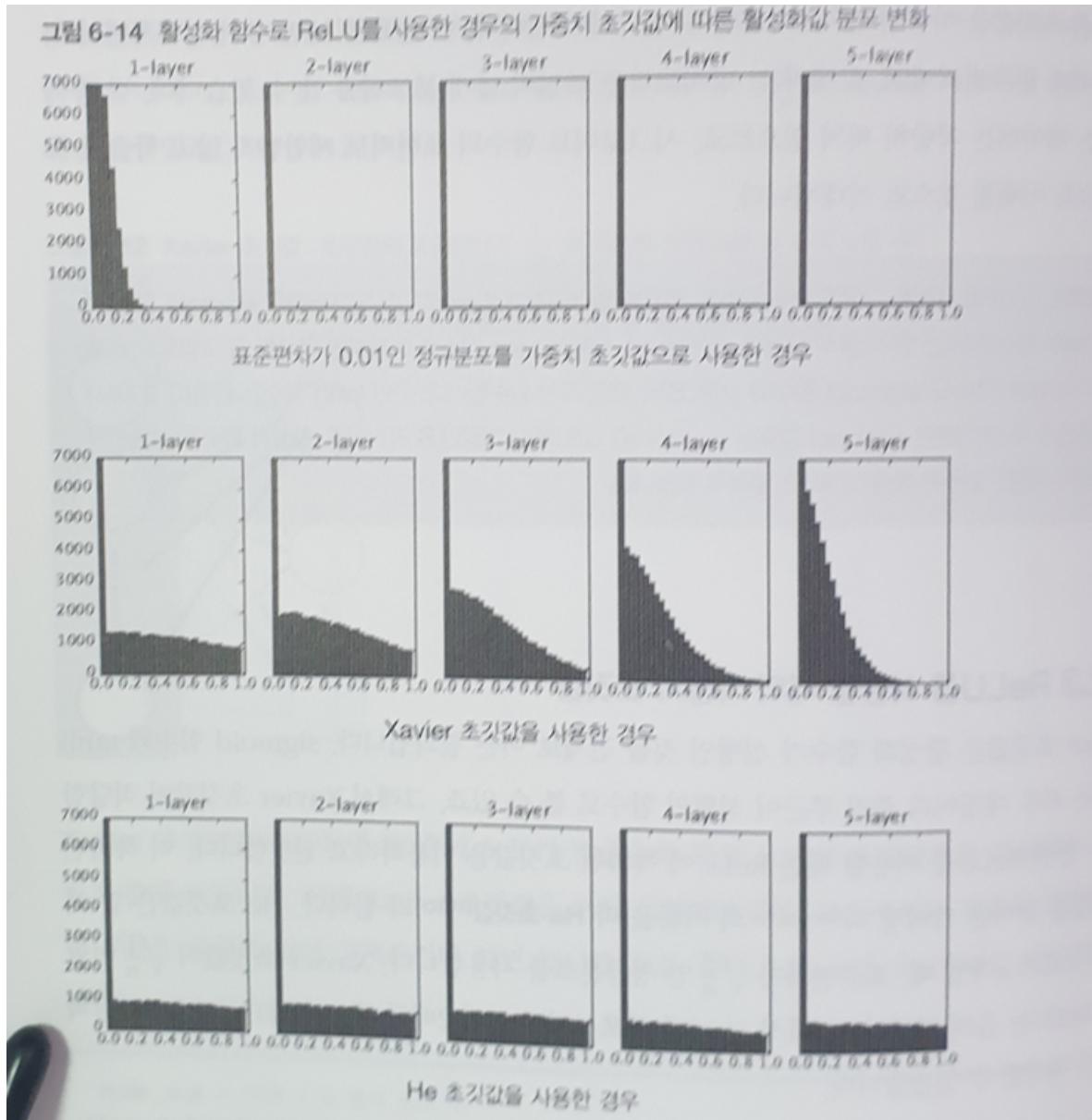
ReLU함수 이용시



6.2.3 ReLU을 사용할 때의 가중치 초기값

Xavier 초기값은 활성화 함수가 선형인 것을 전제로 이끈 결과이다. sigmoid 함수와 tanh 함수는 좌우 대칭이라 중앙 부근이 선형인 함수로 볼 수 있다. 이에 Xavier 초기값이 적당하다. 반면 ReLU 함수를 이용할 때는 ReLU에 특화된 초기값을 이용하라고 권장한다. 이 특화된 초기값을 찾아낸 사람의 이름을 따, 이를 **He 초기값**이라 한다. He 초기값은 앞 계층의 노드가 n 개 일 때, 표준편차가 $\sqrt{\frac{2}{n}}$ 인 정규분포를 사용

한다. Xavier 초깃값은 $\sqrt{\frac{1}{n}}$ 이었다. 이는 음의 영역이 0이라서 더 넓게 분포시키기 위해 2배의 계수가 필요하다고 (직감적으로) 해석할 수 있겠다.



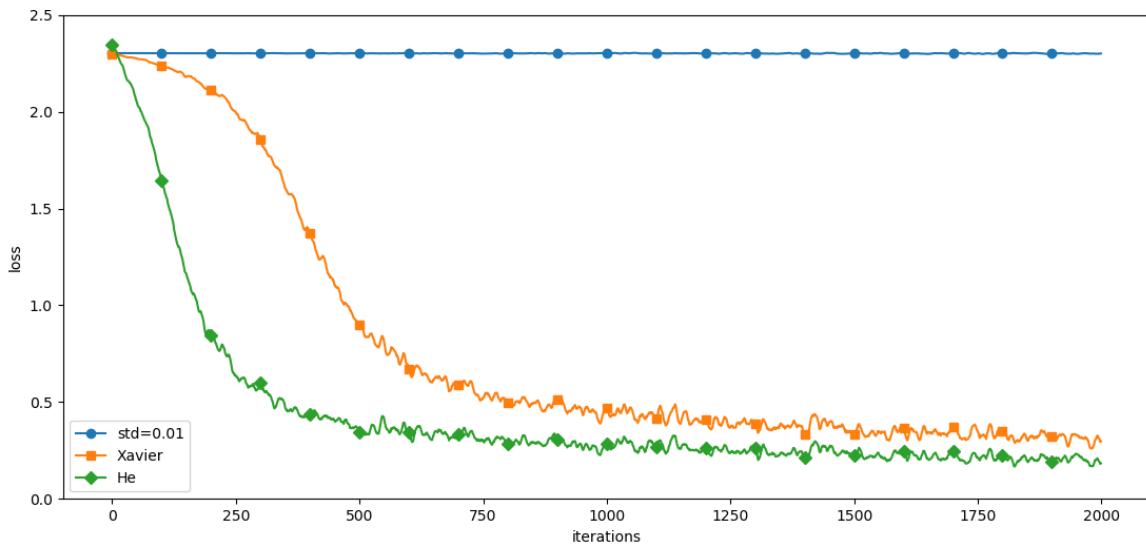
결과를 보면 $std = 0.01$ 일 때의 각 층의 활성화값들은 아주 작은 값들이다. 신경망에 아주 작은 데이터가 흐른다는 것은 역전파 때 가중치의 기울기 역시 작아진다는 뜻이다. 이는 중대한 문제이며, 실제로도 학습이 거의 이뤄지지 않을 것이다.

이어서 Xavier 초기값 결과를 보면 이쪽은 층이 깊어지면서 치우침이 조금씩 커진다. 실제로 층이 깊어지면 활성화값들의 치우침도 커지고, 학습할 때 '기울기 소실' 문제를 일으킨다.

마지막으로 He 초기값은 모든 층에서 균일하게 분포되어 있다. 층이 깊어져도 분포가 균일하게 유지되기 때문에 역전파 때도 적절한 값이 나올 것으로 기대된다.

본 실험 결과를 바탕으로, 활성화 함수로 ReLU를 사용할 때는 He 초기값을, sigmoid 나 tanh 등의 S자 모양 곡선일 때는 Xavier 초기값을 쓰겠다.

6.2.4 MNIST 데이터셋으로 본 가중치 초기값 비교



이 실험은 층별 뉴런 수가 100개인 5층 신경망에서 활성화 함수로 ReLU를 사용했다. 위 그림에서 보듯, $std = 0.01$ 일 때는 학습이 전혀 이뤄지지 않는다. 앞서 활성화값의 분포에서 본 것처럼 순전파 때 너무 작은 값(0 근처로 밀집한 데이터)이 흐르기 때문이다. 그로 인해 역전파 때의 기울기도 작아져 가중치가 거의 갱신되지 않는 것이다. 반대로 Xavier와 He 초기값의 경우는 학습이 순조롭게 이뤄지고 있다. 다만 학습 진도는 He 초기값 쪽이 더 빠르다.

지금까지 살펴봤듯, 가중치의 초기값은 신경망 학습에 있어서 매우 중요하다. 가중치의 초기값에 따라 신경망 학습의 성패가 갈리는 경우도 많다.

6.3 배치 정규화

앞 절에서는 각 층의 활성화값 분포를 관찰해보며, 가중치의 초기값을 적절히 설정하면 각 층의 활성화값 분포가 적당히 퍼지면서 학습이 원활하게 수행됨을 배웠다. 그렇다면 각 층이 활성화를 적당히 퍼뜨리도록 '강제'해보면 어떨까? 실은 **배치 정규화**가 그런 아이디어에서 출발한 방법이다.

6.3.1 배치 정규화 알고리즘

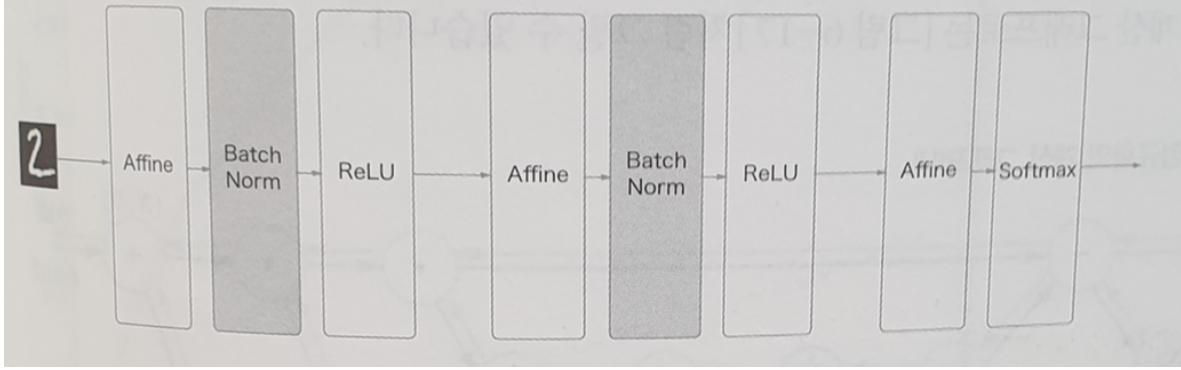
배치 정규화는 2015년에 제안된 방법이다. 배치 정규화는 아직 세상에 나온 지 얼마 안된 기법임에도 많은 연구자와 기술자가 즐겨 사용한다. 실제로 기계학습 콘테스트의 결과를 보면 이 배치 정규화를 사용하여 뛰어난 결과를 달성한 예가 많다. 배치 정규화의 특징은 다음과 같다.

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

딥러닝의 학습시간이 길다는 것을 생각하면 첫 번째 이점은 아주 반가운 일이다. 초기값에 크게 신경 쓸 필요 없고, 오버피팅 억제 효과가 있다는 점도 딥러닝 학습의 두통거리를 덜어준다.

배치 정규화의 기본 아이디어는 앞에서 말했듯 각 층에서의 활성화값이 적당히 분포되도록 조정하는 것이다. 데이터 분포를 정규화하는 '배치 정규화 계층'을 신경망에 삽입한다.

그림 6-16 배치 정규화를 사용한 신경망의 예



배치 정규화는 그 이름과 같이 학습시 미니배치를 단위로 정규화한다. 구체적으로는 데이터의 분포가 평균이 0, 분산이 1이 되도록 정규화한다. 수식으로는 다음과 같다.

$$\begin{aligned}\mu_B &\leftarrow \frac{1}{m} \sum_{i=1}^m x_i \\ \sigma_B^2 &\leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2 \\ \hat{x}_i &\leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}\end{aligned}$$

식 6.7

여기에는 미니배치 $B = \{x_1, x_2, x_3, \dots, x_m\}$ 이라는 m 개의 입력 데이터의 집합에 대해 평균 μ_B 와 분산 σ_B^2 을 구한다. 그리고 입력 데이터를 평균이 0, 분산이 1이 되게 (적절한 분포가 되게) 정규화한다. ϵ 은 0으로 나누는 사태를 방지한다.

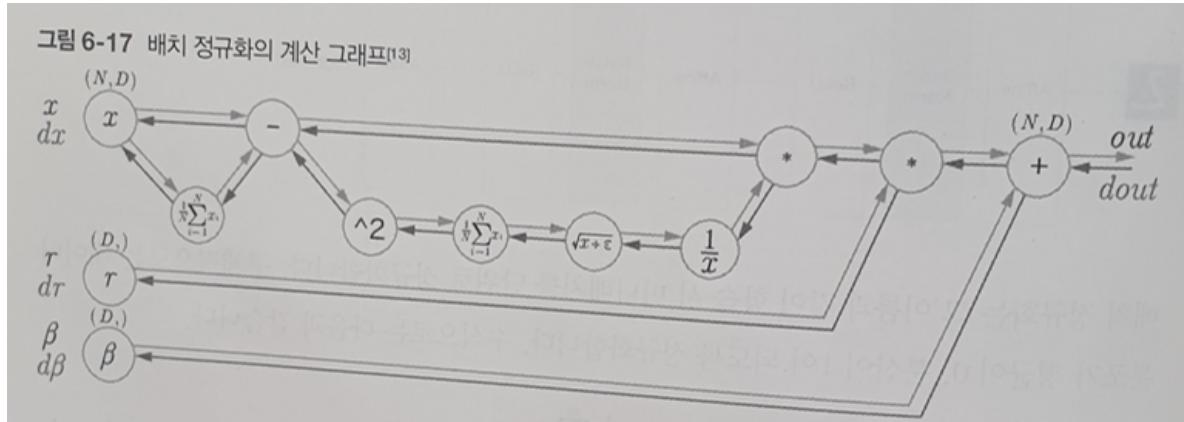
식 6.7은 단순히 미니배치 입력 데이터 $\{x_1, x_2, x_3, \dots, x_m\}$ 을 평균 0, 분산 1인 데이터 $\{\hat{x}_1, \hat{x}_2, \hat{x}_3, \dots, \hat{x}_m\}$ 으로 변환하는 일을 한다. 이 처리를 활성화 함수의 앞(혹은 뒤)에 삽입*함으로써 데이터 분포가 덜 치우치게 할 수 있다.

또, 배치 정규화 계층마다 이 정규화된 데이터에 고유한 확대와 이동 변환을 수행한다. 수식으로는 다음과 같다.

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

이 식에서 γ 가 확대를, β 가 이동을 담당한다. 두 값은 처음에는 $\gamma = 1, \beta = 0$ 부터 시작하고, 학습하면서 적합한 값으로 조정해나간다.

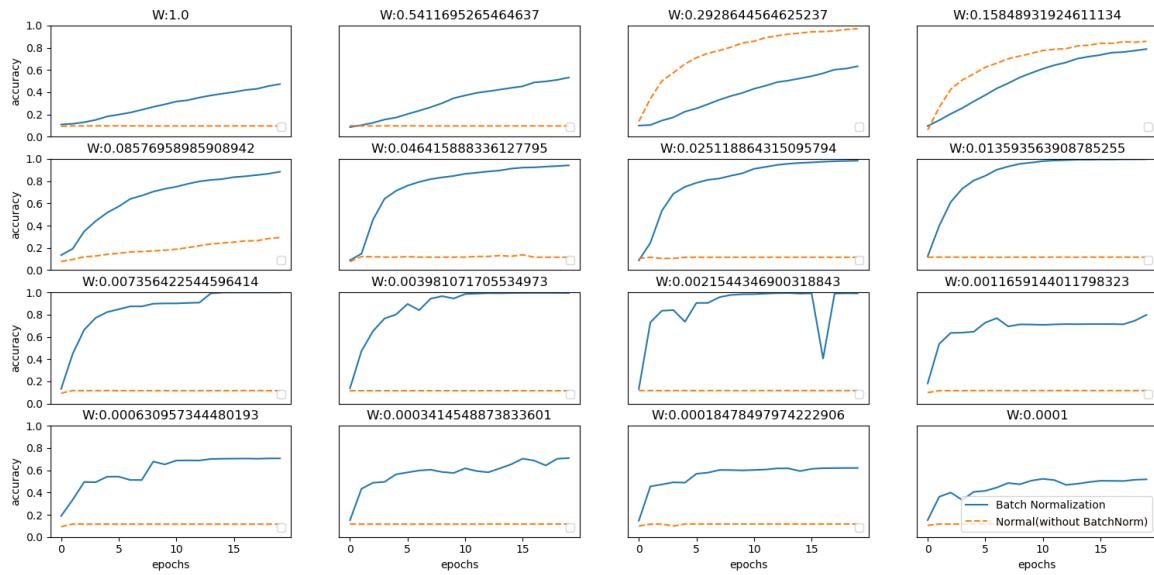
이상이 배치 정규화의 알고리즘이다. 이 알고리즘이 신경망에서 순전파 때 적용된다. 이를 5장에서 설명한 계산 그래프로는 다음처럼 그릴 수 있다.



배치 정규화의 역전파 유도는 다소 복잡하므로 여기서 설명하지는 않겠다.

6.3.2 배치 정규화의 효과

그럼 배치 정규화 계층을 사용한 실험을 해보자. 우선은 MNIST 데이터셋을 사용하여 배치 정규화 계층을 사용할 때와 사용하지 않을 때의 학습 진도가 어떻게 달라지는지를 보겠다.



다음과 같이 배치 정규화가 학습을 빨리 진전시키고 있다. 계속해서 초기값 분포를 다양하게 줘가며 학습 진행이 어떻게 달라지는지 보겠다. 위 그림은 가중치 초기값의 표준편차를 다양하게 바꿔가며 학습 경과를 관찰한 그래프다.

실선이 배치정규화를 사용한 경우, 점선이 그렇지 않은 경우를 표시한 것이다.

거의 모든 경우에서 배치 정규화를 사용할 때의 학습 진도가 빠른 것으로 나타난다. 실제로 배치 정규화를 이용하지 않는 경우엔 초기값이 잘 분포되어 있지 않으면 학습이 전혀 진행되지 않는 모습도 확인할 수 있다.

지금까지 살펴본 것처럼 배치 정규화를 사용하면 학습이 빨라지며, 가중치 초기값에 크게 의존하지 않아도 된다. 배치 정규화는 이처럼 장점이 많으니 앞으로 다양한 분야에서 활약할 것이다.

6.4 바른 학습을 위해

기계학습에서는 **오버피팅**이 문제가 되는 일이 많다. 오버피팅이란 신경망이 훈련 데이터에만 지나치게 적응되어 그 외의 데이터에는 제대로 대응하지 못하는 상태를 말한다. 기계학습은 범용 성능을 지향한다. 훈련 데이터에는 포함되지 않는, 아직 보지 못한 데이터가 주어져도 바르게 식별해내는 모델이 바람직하다. 복잡하고 표현력이 높은 모델을 만들 수는 있지만, 그만큼 오버피팅을 억제하는 기술이 중요해지는 것이다.

6.4.1 오버피팅

오버피팅은 주로 다음의 두 경우에 일어난다.

- 매개변수가 많고 표현력이 높은 모델
- 훈련 데이터가 적음

이번 절에서는 이 두 요건을 일부러 총족하여 오버피팅을 일으켜보겠다. 그러기 위해 본래 60,000개의 MNIST 데이터셋의 훈련 데이터 중 300개만 사용하고, 7층 네트워크를 사용해 네트워크의 복잡성을 높이겠다. 각 층의 뉴런 100개, 활성화 함수 ReLU를 사용한다. 여기에서는 실험에 필요한 코드를 발췌해 설명하겠다.

```
(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]
```

이어서 훈련을 수행하는 코드다. 지금까지의 코드와 같지만 에폭마다 모든 훈련 데이터와 모든 시험 데이터 각각에서 정확도를 산출한다.

```
In [3]: %run overfit_weight_decay.py
epoch:0, train acc:0.11666666666666667, test acc:0.1001
epoch:1, train acc:0.1333333333333333, test acc:0.1041
epoch:2, train acc:0.1433333333333334, test acc:0.1126
epoch:3, train acc:0.17666666666666667, test acc:0.1258
epoch:4, train acc:0.18666666666666668, test acc:0.1437
epoch:5, train acc:0.21, test acc:0.1539
epoch:6, train acc:0.2233333333333333, test acc:0.1671
epoch:7, train acc:0.25, test acc:0.179
```

이어서 훈련을 수행하는 코드다. 지금까지의 코드와 같지만 에폭마다 모든 훈련 데이터와 모든 시험 데이터 각각에서 정확도를 산출한다.

```
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 생성

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

```

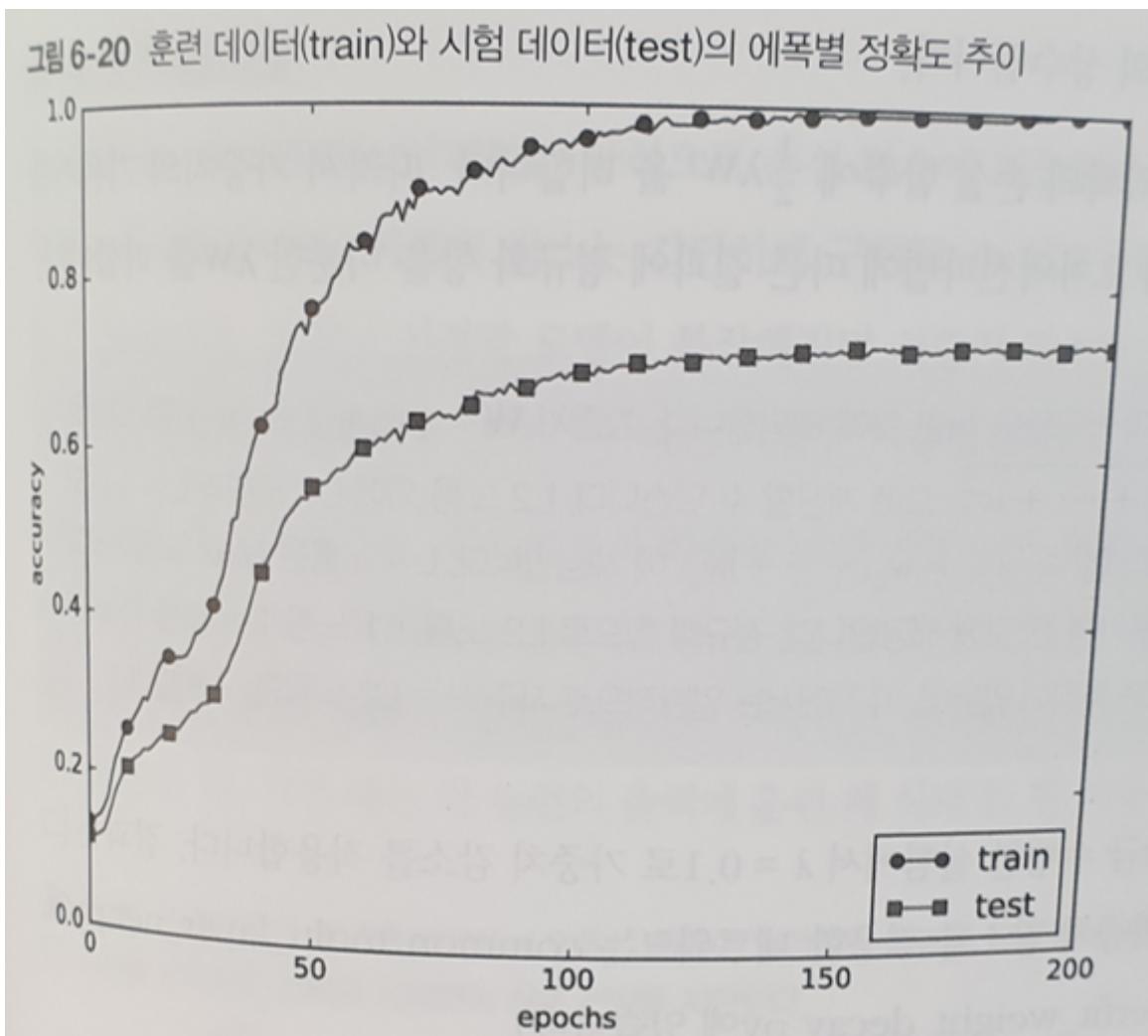
train_acc_list와 test_acc_list에는 에폭 단위(모든 훈련 데이터를 한 번씩 본 단위)의 정확도를 저장한다. 이에 대한 결과는 다음과 같다.

훈련 데이터를 사용하여 측정한 정확도는 100에폭을 지나는 무렵부터 거의 100%이다. 그러나 시험 데이터에 대해서는 큰 차이를 보인다. 이처럼 정확도가 크게 벌어지는 것은 훈련 데이터에만 적용해버린 결과이다. 훈련 때 사용하지 않은 범용 데이터(시험 데이터)에는 제대로 대응하지 못하는 것을 이 그래프에서 확인할 수 있다.

6.4.2 가중치 감소

오버피팅 억제용으로 예로부터 많이 이용해온 방법 중, **가중치 감소**라는 것이 있다. 이는 학습 과정에서 큰 가중치에 대해서는 그에 상응하는 큰 페널티를 부과하여 오버피팅을 억제하는 방법이다. 원래 오버피팅은 가중치 매개변수의 값이 커서 발생하는 경우가 많기 때문이다.

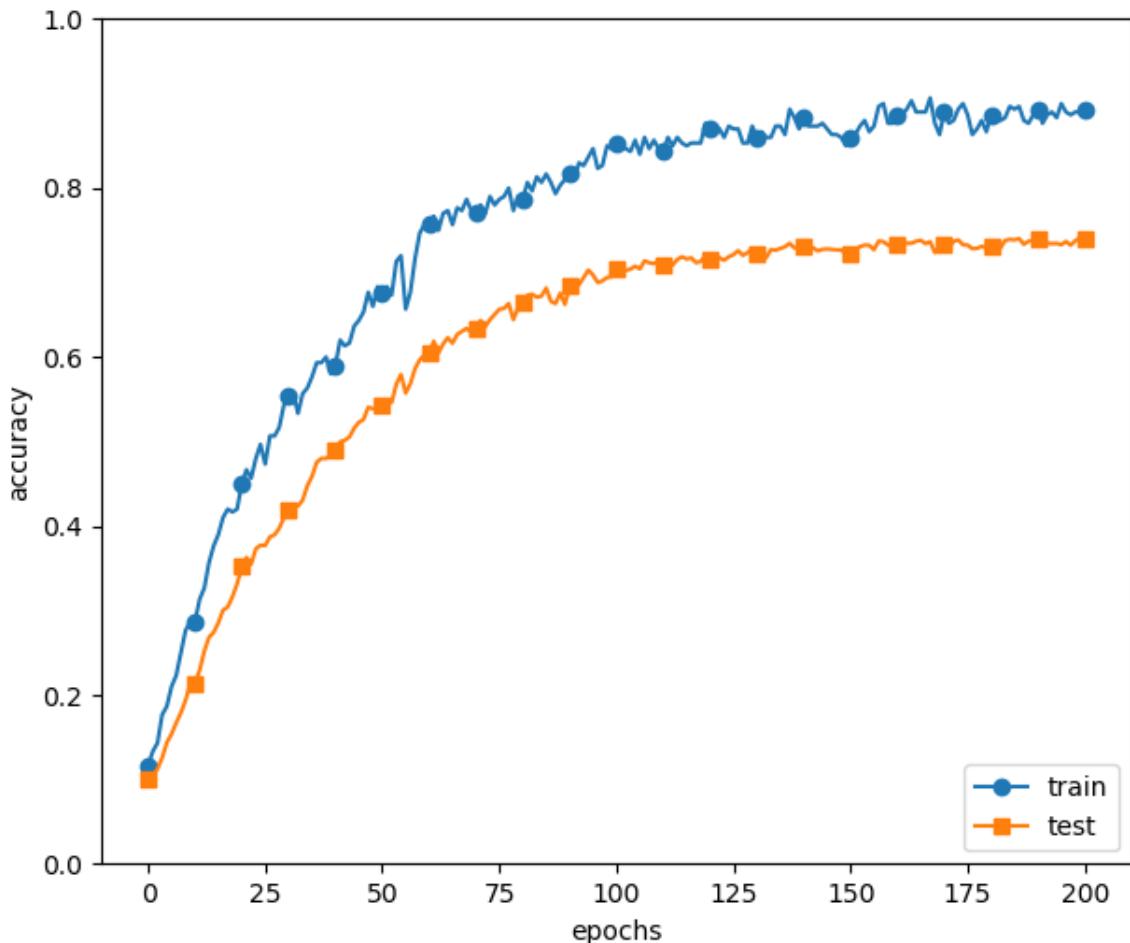
자 복습해보자. 신경망 학습의 목적은 손실 함수의 값을 줄이는 것이다. 예를 들어 가중치의 제곱 노름을 손실 함수에 더한다. 그러면 가중치가 커지는 것을 억제할 수 있다. 가중치를 \mathbf{W} 라 하면 L2 노름에 따른 가중치 감소는 $\frac{1}{2}\lambda W^2$ 이 되고 이 $\frac{1}{2}\lambda W^2$ 을 손실 함수에 더한다. 여기서 λ 는 정규화의 세기를 조절하는 하이퍼파라미터이다. λ 는 정규화의 세기를 조절하는 하이퍼파라미터이다. λ 를 크게 설정할수록 큰 가중치에 대한 페널티가 커진다. 또 $\frac{1}{2}\lambda W^2$ 의 앞쪽 $\frac{1}{2}$ 은 $\frac{1}{2}\lambda W^2$ 의 미분 결과인 λW 를 조정하는 역할의 상수이다.



가중치 감소는 모든 가중치 각각의 손실 함수에 $\frac{1}{2}\lambda W^2$ 을 더한다. 따라서 가중치의 기울기를 구하는 계산에서 그동안의 오차역전파법에 따른 결과에 정규화 항을 미분한 λW 를 더한다.

Note L2 노름은 각 원소의 제곱들을 더한 것에 해당한다. 가중치 $W = (w_1, w_2, \dots, w_n)$ 이 있다면, L2 노름에서는 $\sqrt{w_1^2 + w_2^2 + \dots + w_n^2}$ 으로 계산할 수 있다. L2 노름 외에 L1 노름과 L_∞ 노름도 있다. L1 노름은 절댓값의 합, 즉 $|w_1| + |w_2| + \dots + |w_n|$ 에 해당한다. L_∞ 은 Max 노름이라고도 하며, 각 원소의 절댓값 중 가장 큰 것에 해당한다. 정규화 항으로 L2 노름, L1 노름, L_∞ 노름 중 어떤 것도 사용할 수 있다. 각자 특징이 있는데, 이 책에서는 일반적으로 자주 쓰는 L2 노름만 구현한다.

그럼 실험해보자. 방금 수행한 실험에서 $\lambda = 0.1$ 로 가중치 감소를 적용한다.

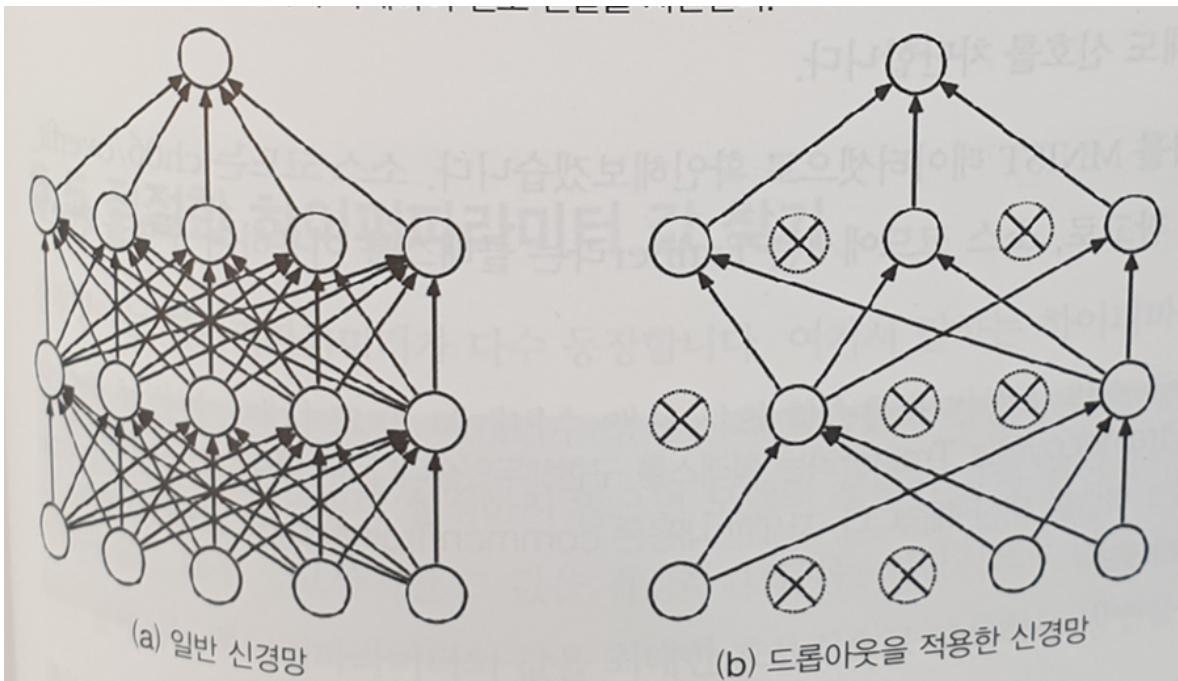


위 그림과 같이 훈련 데이터에 대한 정확도와 시험 데이터에 대한 정확도에는 여전히 차이가 있지만, 가중치 감소를 이용하지 않은 저 위 그림과 비교하면 그 차이가 줄었다. 다시 말해 오버피팅이 억제됐다는 소리이다. 그리고 앞서와 달리 훈련 데이터에 대한 정확도가 100%(1.0)에 도달하지 못한 점도 주목해야 겠다.

6.4.3 드롭아웃

앞 절에서는 오버피팅을 억제하는 방식으로 손실 함수에 가중치의 L2 노름을 더한 가중치 감소 방법을 설명했다. 가중치 감소는 간단하게 구현할 수 있고 어느 정도 지나친 학습을 억제할 수 있다. 그러나 신경망 모델이 복잡해지면 가중치 감소만으로는 대응하기 어려워진다. 이럴 때는 흔히 **드롭아웃**이라는 기법을 이용한다.

드롭아웃은 뉴런을 임의로 삭제하면서 학습하는 방법이다. 한편 때 은닉층의 뉴런을 무작위로 골라 삭제 한다. 삭제된 뉴런은 다음 그림과 같이 신호를 전달하지 않는다. 훈련 때는 데이터를 흘릴 때마다 삭제할 뉴런을 무작위로 선택하고, 시험 때는 모든 뉴런에 신호를 전달한다. 단, 시험 때는 각 뉴런의 출력에 훈련 때 삭제 안한 비율을 곱하여 출력한다.



이제 드롭아웃을 구현할 차례다. 다음 코드는 되도록 이해하기 쉽게 구현한 것이다. 순전파를 담당하는 forward 메서드에서는 훈련 때(`train_flg = True`일 때) 만 잘 계산해두면 시험 때는 단순히 데이터를 흘리기만 하면 된다. 삭제 안한 비율은 곱하지 않아도 좋다. 실제 딥러닝 프레임워크들도 비율을 곱하지 않는다.

```

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_ratio
            return x * self.mask
        else:
            return x*(1.0-self.dropout_ratio)

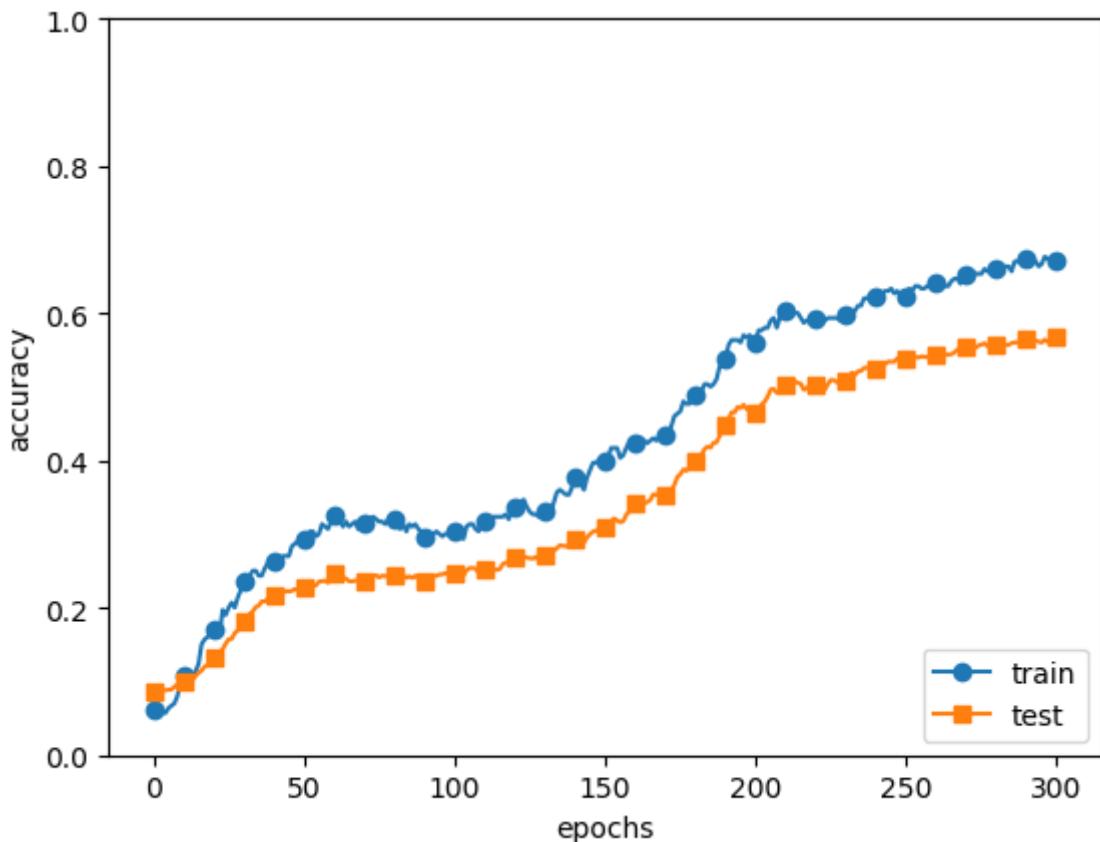
    def backward(self, dout):
        return dout * self.mask

```

여기서의 핵심은 훈련 시에는 순전파 때마다 `self.mask`에 삭제할 뉴런을 `False`로 표시한다는 것이다. `self.mask` 는 `x`와 형상이 같은 배열을 무작위로 생성하고, 그 값이 `dropout_ratio` 보다 큰 원소만 `True`로 설정한다. 역전파 때의 동작은 ReLU와 같다. 즉, 순전파 때 신호를 통과시키는 뉴런은 역전파 때도 신호를 그대로 통과시키고, 순전파 때 통과시키지 않은 뉴런은 역전파 때도 신호를 차단한다.

그럼 드롭아웃의 효과를 MNIST 데이터셋으로 확인해보겠다.

이 실험은 앞의 실험과 마찬가지로 7층 네트워크(각 층의 뉴런 수는 100개, 활성화 함수는 ReLU)를 써서 진행했다.



그림과 같이 드롭아웃을 적용하니 훈련 데이터와 시험 데이터에 대한 정확도 차이가 줄었다. 또, 훈련 데이터에 대한 정확도가 100%에 도달하지 않게 됐다. 이처럼 드롭아웃을 이용하면 표현력을 높이면서 오버피팅을 억제할 수 있다.

Note 기계학습에서는 양상을 학습을 애용한다. 양상을 학습은 개별적으로 학습시킨 여러 모델의 출력을 평균 내어 추론하는 방식이다. 신경망의 맥락에서 얘기하면, 가령 같은(혹은 비슷한) 구조의 네트워크를 5개 준비하여 따로따로 학습시키고, 시험 때는 그 5개의 출력을 평균 내어 답하는 것이다. 양상을 학습을 수행하면 신경망의 정확도가 몇% 정도 개선된다는 것이 실험적으로 알려져 있다. 양상을 학습은 드롭아웃과 밀접하다. 드롭아웃이 학습 때 뉴런을 무작위로 삭제하는 행위를 매번 다른 모델을 학습시키는 것으로 해석할 수 있기 때문이다. 그리고 추론 때는 뉴런의 출력에 삭제한 비율(이를 테면 0.5 등)을 곱함으로써 양상을 학습에서 여러 모델의 평균을 내는 것과 같은 효과를 얻는 것이다. 즉, 드롭아웃은 양상을 학습과 같은 효과를(대략) 하나의 네트워크로 구현했다고 생각할 수 있다.

6.3 적절한 하이퍼파라미터 값 찾기

신경망에는 하이퍼파라미터가 다수 등장한다. 여기서 말하는 하이퍼파라미터는, 예를 들어 각 층의 뉴런 수, 배치 크기, 매개변수 갱신 시의 학습률과 가중치 감소 등이다. 이러한 하이퍼파라미터의 값을 적절히 설정하지 않으면 모델의 성능이 크게 떨어지기도 한다. 하이퍼파라미터의 값은 매우 중요하지만 그 값을 결정하기까지는 일반적으로 많은 시행착오로 겪는다. 이번 절에서는 하이퍼파라미터의 값을 최대한 효율적으로 탐색하는 방법을 설명한다.

6.3.1 검증 데이터

지금까지는 데이터셋을 훈련 데이터와 시험 데이터, 두 가지로 분리해 이용했다. 훈련 데이터로는 학습을 하고, 시험 데이터로는 범용 성능을 평가했다. 그렇게 해서 훈련 데이터에만 지나치게 적응되어 있지 않은지(오버피팅한 건 아닌지), 그리고 범용 성능은 어느 정도 인지 같은 것을 평가할 수 있었다.

앞으로 하이퍼파라미터를 다양한 값으로 설정하고 검증할 텐데, 여기서 주의할 점은 하이퍼파라미터의 성능을 평가할 때는 시험 데이터를 사용해서는 안된다는 것이다. 매우 중요하지만 놓치기 쉬운 포인트다.

왜?? 그것은 시험 데이터를 사용하여 하이퍼파라미터를 조정하면 하이퍼파라미터 값이 시험 데이터에 오버피팅되기 때문이다. 바꿔 말하면, 하이퍼파라미터 값의 '좋음'을 시험 데이터로 확인하게 되므로 하이퍼파라미터의 값이 시험 데이터에만 적합하도록 조정되어 버린다. 이에 다른 데이터에는 적용하지 못하니 범용 성능이 떨어지는 모델이 될지도 모른다.

이에 하이퍼파라미터를 조정할 때는 하이퍼파라미터 전용 확인 데이터가 필요하다. 하이퍼파라미터 조정용 데이터를 일반적으로 **검증 데이터(validation data)**라고 부른다. 하이퍼파라미터의 적절성을 평가하는 데이터인 셈이다.

Note 훈련 데이터는 매개변수(가중치와 편향)의 학습에 이용하고, 검증 데이터는 하이퍼파라미터의 성능을 평가하는데 이용한다. 시험 데이터는 범용 성능을 확인하기 위해 마지막에 (이성적으로는 한 번만) 이용한다.

- 훈련 데이터 : 매개변수 학습
- 검증 데이터 : 하이퍼파라미터 성능 평가
- 시험 데이터 : 신경망의 범용 선응 평가

데이터셋에 따라서는 훈련 데이터, 검증 데이터, 시험 데이터로 미리 분리해둔 것도 있다. 하지만 MNIST 데이터셋은 훈련 데이터와 시험 데이터로만 분리해뒀다. 이런 경우엔 (필요하면) 사용자가 직접 데이터를 분리해야겠다. MNIST 데이터셋에서 검증 데이터를 얻는 가장 간단한 방법은 훈련 데이터 중 20% 정도를 검증 데이터로 먼저 분리하는 것이다. 코드로는 다음과 같다.

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

# 훈련 데이터를 뒤 섞는다.
x_train, t_train = shuffle_dataset(x_train, x_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:]
```

이 코드는 훈련 데이터로 분리하기 전에 입력 데이터와 정답 레이블을 뒤섞는다. 데이터 셋 안의 데이터가 치우쳐 있을지도 모르기 때문이다(예컨대 숫자 '0'에서 '9'까지 순서대로 정렬되어 있을 수 있겠다). 참고로 여기에서 사용한 `shuffle_dataset` 함수는 `np.random.shuffle`을 이용한 것이다.

이어서 검증 데이터를 사용하여 하이퍼파라미터를 최적화하는 기법을 살펴보자.

6.5.2 하이퍼파라미터 최적화

하이퍼파라미터를 최적화할 때의 핵심은 하이퍼파라미터의 '최적 값'이 존재하는 범위를 조금씩 줄여간다는 것이다. 범위를 조금씩 줄이려면 우선 대략적인 범위를 설정하고 그 범위에서 무작위로 하이퍼파라미터 값을 골라낸 (샘플링) 후, 그 값으로 정확도를 평가한다.

정확도를 잘 살피면서 이 작업을 여러 번 반복하며 하이퍼파라미터의 '최적 값'의 범위를 좁혀가는 것이다.

Note 신경망의 하이퍼파라미터 최적화에서는 그리드 서치 같은 규칙적인 탐색보다는 무작위로 샘플링 해 참색하는 편이 좋은 결과를 낸다고 알려져 있다. 이는 최종 정확도에 미치닌 영향력이 하이퍼파라미터마다 다르기 때문이다.

하이퍼파라미터의 범위는 '대략적으로' 지정하는 것이 효과적이다. 실제로도 0.001에서 1,000 사이(10^{-3} 에서 10^3)와 같이 '10의 거듭제곱' 단위로 범위를 지정한다. 이를 '로그 스케일로 지정'한다고 한다.

하이퍼파라미터를 최적화할 때는 딥러닝 학습에는 오랜 시간(예컨대 며칠이나 몇 주 이상)이 걸린다는 점을 기억해야 한다. 따라서 나쁠 듯한 값은 일찍 포기하는 것이 좋다. 이에 학습을 위한 에폭을 작게 하여, 1회 평가에 걸리는 시간을 단축하는 것이 효과적이다.

이상이 하이퍼파라미터의 최적화이다. 지금까지의 이야기를 정리하면 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(에폭은 작게 설정한다).

- **3단계**

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

이상을 반복하여 하이퍼파라미터의 범위를 좁혀가고, 어느 정도 좁아지면 그 압축한 범위에서 값을 하나 골라낸다. 이것이 하이퍼파라미터를 최적화하는 하나의 방법이다.

Note 여기에서 설명한 하이퍼파라미터 최적화 방법은 실용적인 방법이다. 하지만 과학이라기보다는 다른 향수 수행자의 '지혜'와 '직관'에 의존한다는 느낌이 든다. 좀 더 세련된 기법을 원한다면 베이즈 최적화를 사용할 수도 있겠다. 베이즈 최적화는 베이즈 정리를 중심으로 한 수학이론을 구사하여 더 엄밀하고 효율적으로 최적화를 수행한다.

6.5.3 하이퍼파라미터 최적화 구현하기

그럼 MNIST 데이터셋을 사용하여 하이퍼파라미터를 최적화해보기로 하자. 여기에서는 학습률과 가중치 감소의 세기를 조절하는 계수(가중치 감소 계수)를 탐색하는 문제를 풀어보겠다.

앞에서 말한 대로, 하이퍼파라미터의 검증은 그 값을 $0.001\sim1,000$ 사이 같은 로그 스케일 범위에서 무작위로 추출해 수행한다. 이를 파이썬 코드로는 $10^{**\text{np.random.uniform}(-3,3)}$ 처럼 작성할 수 있다. 이 예에서는 가중치 감소 계수를 $10^{-8}\sim10^{-4}$, 학습률을 $10^{-6}\sim10^{-2}$ 범위부터 시작한다. 이 경우 하이퍼파라미터의 무작위 추출 코드는 다음과 같다.

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

이렇게 무작위로 추출한 값을 사용하여 학습을 수행한다. 그 후에는 여러 차례 다양한 하이퍼파라미터 값을 반복하며 신경망에 좋을 것 같은 값이 어디에 존재하는지 관찰한다.

가중치 감소 계수의 범위를 $10^{-8}\sim10^{-4}$, 학습률을 $10^{-6}\sim10^{-2}$ 로 하여 실험하면 결과는 다음처럼 된다.

```
(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, epochs=50):
    network = MultiLayerNet(input_size=784, hidden_size_list=[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=epochs, mini_batch_size=100,
                      optimizer='sgd', optimizer_param={'lr': lr}, verbose=False)
    trainer.train()

    return trainer.test_acc_list, trainer.train_acc_list

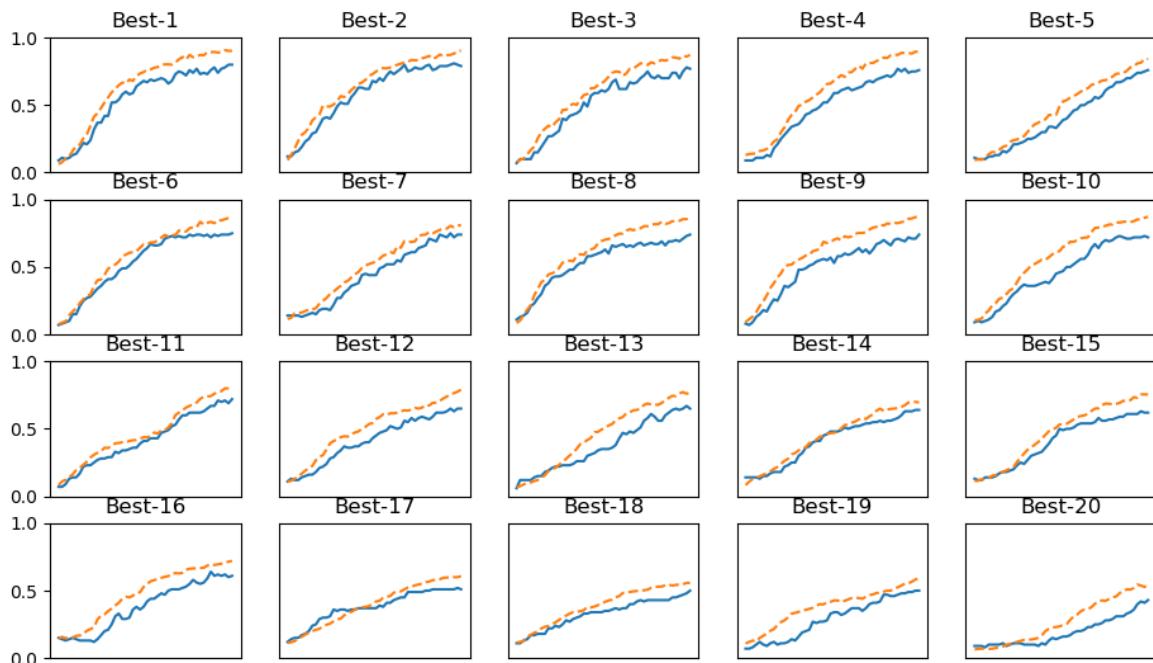
```

```

# 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

```



위 그림은 검증 데이터의 학습 추이를 정확도가 높은 순서로 나열했다. 이를 보면 'Best-5' 정도까지는 학습이 순조롭게 진행되고 있다. 이를 바탕으로 'Best-5' 까지의 하이퍼파라미터의 값(학습률과 가중치 감소 계수)을 살펴보겠다.

```
===== Hyper-Parameter Optimization Result =====
Best-1(val acc:0.8) | lr:0.008146705715689243, weight decay:5.933288788431116e-05
Best-2(val acc:0.79) | lr:0.008739405846252032, weight decay:1.743536076667593e-05
Best-3(val acc:0.77) | lr:0.007052130590448214, weight decay:5.2239282026736614e-08
Best-4(val acc:0.76) | lr:0.007915696923845313, weight decay:6.407451574171807e-08
Best-5(val acc:0.76) | lr:0.00825610807484735, weight decay:1.626197018774021e-06
```

이 결과를 보면 학습이 잘 진행될 때의 학습률은 0.001~0.01, 가중치 감소 계수는 10^{-8} ~ 10^{-6} 정도라는 것을 알 수 있다. 이처럼 잘 될 것 같은 값의 범위를 관찰하고 범위를 좁혀간다. 그런 다음 그 축소된 범위로 똑같은 작업을 반복한다. 이렇게 적절한 값이 위치한 범위로 좁혀가다가 특정 단계에서 최종 하이퍼파라미터 값을 하나 선택한다.

6.6 정리

이번 장에서는 신경망 학습에 중요한 기술 몇 가지를 소개했다. 매개변수 갱신 방법과 가중치의 초기값을 설정하는 방법, 또 배치 정규화와 드롭아웃 등 현대적인 신경망에서 빼놓을 수 없는 기술들이다.

이번 장에서 배운 내용

- 매개변수 갱신 방법에는 경사 하강법(SGD) 외에도 모멘텀, AdaGrad, Adam 등이 있다.
- 가중치 초기값을 정하는 방법은 올바른 학습을 하는 데 매우 중요하다.
- 가중치의 초기값으로는 'Xavier 초기값'과 'He 초기값'이 효과적이다.
- 배치 정규화를 이용하면 학습을 빠르게 진행할 수 있으며, 초기값에 영향을 덜 받는다.
- 오버피팅을 억제하는 정규화 기술로는 가중치 감소와 드롭아웃이 있다.
- 하이퍼파라미터 값 탐색은 최적 값이 존재할 법한 범위를 점차 좁히면서 하는 것이 효과적이다.