Ch 06 학습 관련 기술들

이번 장에서는 신경망 학습의 핵심 개념들인 가중치 매개변수의 최적값을 탐색하는 최적화 방법, 가중치 매개변수 초깃값, 하이퍼파라미터 설정 방법 등에 대해 알아봅시다.

6.1 매개변수 갱신

신경망 학습의 목적은 손실 함수의 값을 최대한 낮추는 가중치 매개변수를 탐색하는 것입니다. 이런 문제를 푸는 것을 최적화(optimization)라 합니다. 점점 신경망이 층을 두껍게 쌓기 시작하면서 매개변수의 수가 엄청나게 많아졌기 때문에 최적화가 중요해졌습니다. 앞에서는 매개변수의 기울기를 구해, 기울어진 방향으로 매개변수를 갱신하는 일을 반복해서 최적의 값에 다가갔습니다. 이는 확률적경사하강법(SGD)이고 보다 똑똑한 방법도 있습니다. 지금부터 다른 최적화 기법에 대해 알아봅시다.

6.1.1 모험가 이야기

책에서 최적화를 해야 하는 상황을 모험가 이야기에 비유했습니다.

*색다른 모험가가 있습니다. 광활한 메마른 산맥을 여행하면서 날마다 깊은 골짜기를 찾아 발걸음을 옮깁니다. 그는 전설에 나오는 세상에서 가장 깊고 낮은 골짜기, ‘깊은 곳’을 찾아가려 합니다. 그것이 그의 여행 목적이죠. 게다가 그는 엄격한 ‘제약’ 2개로 자신을 옭아맸습니다. 하나는 지도를 보지 않을 것, 또 하나는 눈가리개를 쓰는 것입니다. 지도도 없고 보이지도 않으니 가장 낮은 골짜기가 광대한 땅 어디에 있는지 알 도리가 없죠. 그런 혹독한 조건에서 이 모험가는 어떻게 ‘깊은 곳’을 찾을 수 있을까요? 어떻게 걸음을 옮겨야 효율적으로 ‘깊은 곳’을 찾아낼 수 있을까요?*

이 모험가 이야기는 최적 매개변수를 탐색하는 우리의 상황과 같습니다. 이런 ‘제약’이 걸린 상황에서 ‘기울기’는 매우 중요한 단서가 됩니다. 발바닥으로 전해지는 바닥의 기울기를 바탕으로 가장 기울어진 방향으로 가는 것이 SGD의 전략입니다.

6.1.2 확률적 경사 하강법(SGD)

SGD를 복습해 보겠습니다. 수식은 다음과 같습니다.

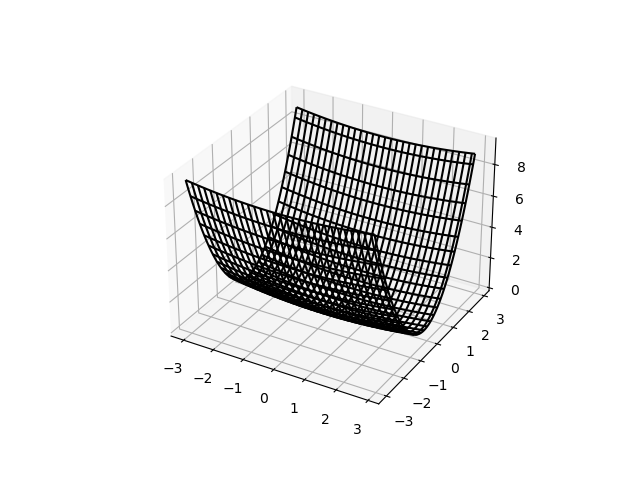
W는 갱신할 가중치 매개변수, 는 학습률, 는 W에 대한 손실함수의 기울기입니다. 이런 SGD를 파이썬으로 구현해보겠습니다.

class SGD:  
 def \_\_init\_\_(self, lr=0.01):  
 self.lr = lr  
  
 def update(self, params, grads):  
 for key in params.key():  
 params[key] -= self.lr \* grads[key]

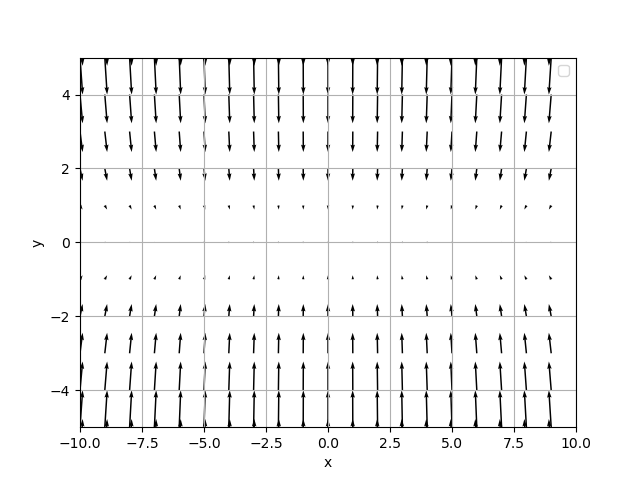
6.1.3 SGD의 단점

SGD는 단순하고 구현이 쉽지만, 문제에 따라 비효율적일 때가 있습니다. 다음 경우를 봅시다.

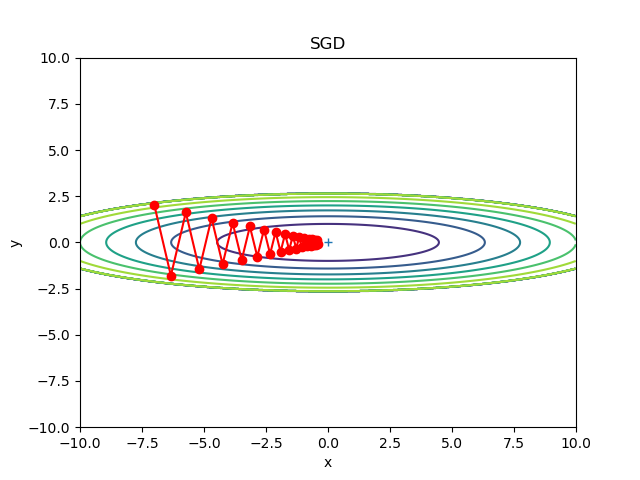
그래프를 그리면 다음과 같이 나옵니다.



이런 그래프에서의 기울기 벡터를 그려보면 다음과 같습니다.



화살표 방향이 낮은 쪽을 가리키고 있지만 최솟값이 되는 곳인 (0, 0) 방향을 가리키지는 못합니다. 이런 함수에서 SGD를 적용하면 다음과 같이 결과가 나옵니다.



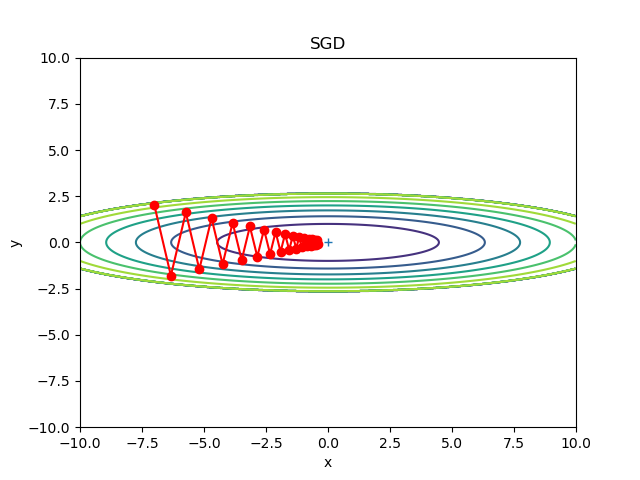
SGD의 움직임을 보면, 비효율적으로 움직입니다. 즉, SGD는 비등방성(anisotropy) 함수 (방향에 따라 성질, 기울기가 달라지는 함수)에서는 탐색 경로가 비효율적입니다. 이런 SGD의 단점을 개선해준 모멘텀, AdaGrad, Adam이라는 세 방법에 대해 알아보겠습니다.

6.1.4 모멘텀

모멘텀(Momentum)은 운동량을 뜻하는 단어로 물리와 관계가 있습니다. 수식으로는 다음과 같습니다.

새로 v라는 변수가 나오는데, 이는 물리에서 말하는 속도(velocity)에 해당합니다. 위 식은 기울기 방향으로 힘을 받아 물체가 가속된다는 물리 법칙을 나타냅니다. av항은 물체가 아무런 힘을 받지 않을 때 속도를 서서히 낮추는 역할을 합니다. 다음은 모멘텀의 구현과 그래프 입니다.

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.key():  
 self.v[key] = self.momentum \* self.v[key] - self.lr \* grads[key]  
 params[key] += self.v[key]



그림에서 볼 수 있듯 모멘텀의 경로는 공이 바닥을 구르듯 움직입니다. SGD와 비교했을 때, ‘지그재그 정도’가 적은 것을 볼 수 있습니다. x축의 힘이 작지만 방향이 변하지 않기 때문에 한 방향으로 일정하게 가속하기 때문입니다. SGD보다 x축 방향으로 빠르게 다가가 지그재그 움직임이 줄어들었습니다.

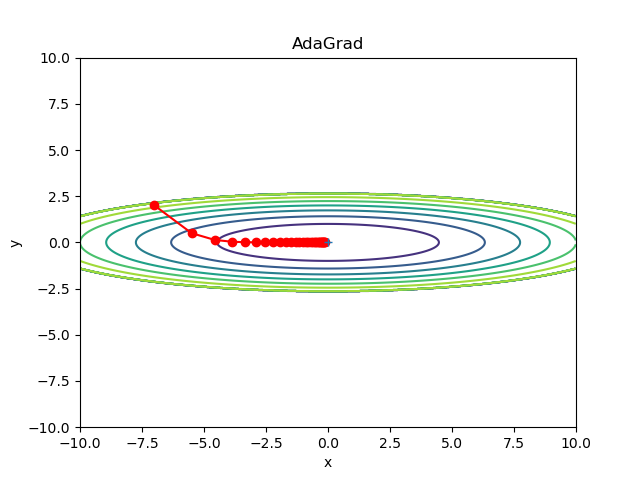
6.1.5 AdaGrad

신경망 학습에서 학습률 값은 중요합니다. 값이 너무 작으면 학습 시간이 줄어들고, 너무 크면 발산하여 학습이 제대로 이뤄지지 않습니다. 이러한 학습률을 정하는 기술에 학습률 감소(learning rate decay)가 있습니다. 학습을 진행하면서 학습률을 점차 줄여가는 방법입니다. 이런 방법을 발전시킨 것이 AdaGrad입니다. AdaGrad는 각각의 매개변수에 맞춤형(Adaptive) 값을 만들어줍니다. 수식으로는 다음과 같습니다.

여기에서는 새로 h라는 변수가 등장합니다. h는 기존 기울기 값을 제곱하여 계속 더해줍니다. 그리고 매개변수 W를 갱신할 때 를 곱해주는데, 이는 매개변수의 원소 중에서 많이 움직인 원소는 학습률이 낮아진다는 뜻이며, 즉 학습률 감소가 매개변수의 원소마다 다르게 적용됨을 뜻합니다. 여기서 주의해야할 AdaGrad의 특징 중에는 학습을 진행할수록 갱신 강도가 점점 약해져 어느 순간 갱신량이 0이 되어버립니다. 이런 문제를 개선한 방법이 RMSProp입니다. RMSProp는 먼 과거의 기울기는 서서히 잊고 새로운 기울기 정보를 크게 반영합니다. 이를 지수이동평균(Exponential Moving Average EMA)라 하며, 과거 기울기의 반영 규모를 기하급수적으로 감소시킵니다. 이제 파이썬으로 AdaGrad를 구현해보겠습니다.

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

마지막 줄에서 1e-7이라는 작은 값을 더하여 0으로 나누는 divide by zero 에러를 막아줍니다. AdaGrad의 그래프는 다음과 같이 나옵니다.



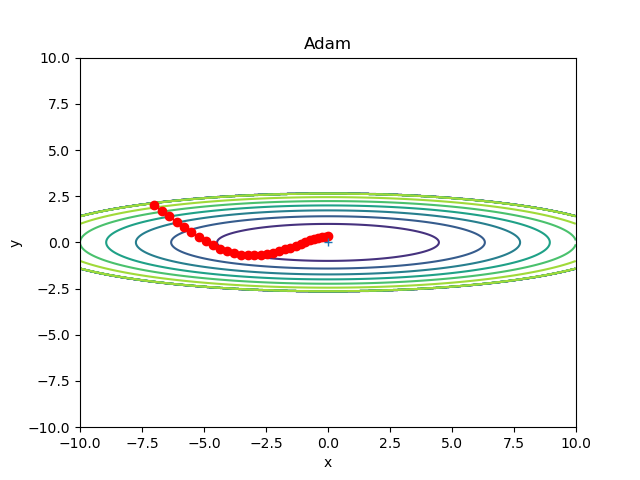
이전의 optimizer들 보다 확실히 빠르게 최솟값에 접근하는 것을 볼 수 있습니다. y축은 기울기가 커서 빠르게 움직이지만, 그 큰 움직임에 비례해 갱신 정도도 큰 폭으로 줄어듭니다.

6.1.6 Adam

모멘텀과 AdaGrad 두 기법을 융합하면 어떻게 될까요? 이런 생각에서 출발한 기법이 Adam입니다. 두 방법을 조합했으니 매개변수 갱신에도 효율적일 것입니다. Adam은 책에서는 자세히 설명되지 않았습니다. 책에서는 ‘편향 보정’에 대해 언급했고 궁금하신 분은 원논문을 참고하라고 되어있습니다. 우선 책에서 제공한 Adam 클래스를 구현해보도록 하겠습니다.

class Adam:  
 def \_\_init\_\_(self, lr=0.01, 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.key():  
 self.m[key] += (1 - self.beta1) \* (grads[key] - self.m[key])  
 self.v[key] += (1 - self.beta2) \* (grads[key]\*\*2 - self.m[key])  
  
 params[key] -= lr\_t \* self.m[key] / (np.sqrt(self.v[key]) + 1e-7)

그래프를 보도록 하겠습니다.



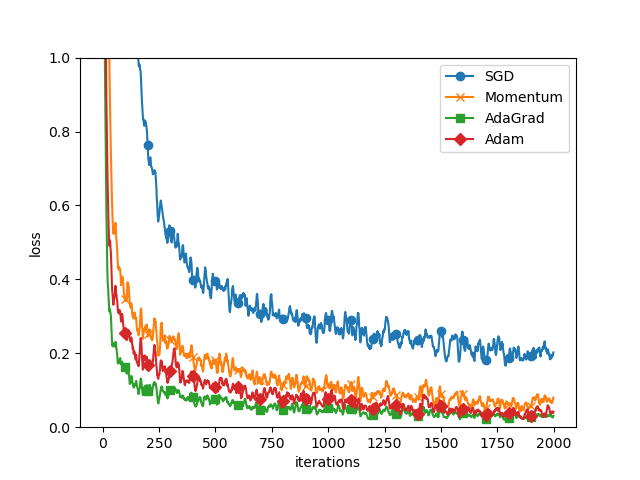
모멘텀과 비슷한 패턴이지만, 지그재그 움직임이 줄어들었습니다.

6.1.7 어느 갱신 방법을 이용할 것인가?

Optimizer에 따른 갱신경로는 모두 달랐습니다. 아직까지 모든 문제에 항상 뛰어난 기법은 존재하지 않기 때문에, 각각의 장단을 고려하여 알맞은 갱신 방법을 사용하는 것이 좋습니다. 요즘에는 많은 사람들이 Adam에 만족하며 쓰는 것 같다고 합니다.

6.1.8 MNIST 데이터셋으로 본 갱신 방법 비교

책의 예제 코드인 optimizer\_compare\_mnist.py를 사용하여 각 방법의 학습 진도가 얼마나 다른 지 보겠습니다.



6.2 가중치의 초깃값

가중치의 초깃값은 신경망 학습에서 중요합니다. 이번 절에서 권장 초깃값에 대해 알아보고 실제 성능을 비교해보도록 하겠습니다.

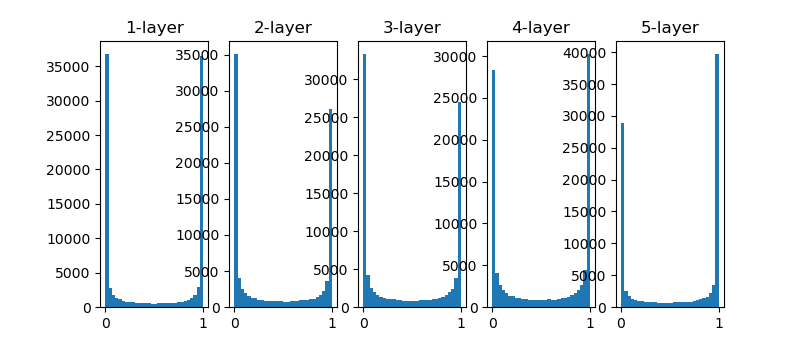
6.2.1 초깃값을 0으로 하면?

지금부터 가중치 감소(weight decay)기법에 대해 알아보도록 하겠습니다. 가중치 감소는 가중치 매개변수의 값이 작아지도록 학습하는 방법입니다. 가중치 값을 작게 하여 오버피팅이 일어나지 않게 하는 것입니다. 가중치를 작게 만들고 싶으면 작은 값(0)에서 시작하는 것은 어떨까요? 가중치를 균일한 값으로 설정하는 것은 좋지 않습니다. 그 이유는 오차역전파법에서 모든 가중치의 값이 똑같이 갱신되기 때문입니다. 이런 ‘가중치가 고르게 되어버리는 상황’을 막으려면 초깃값을 무작위로 설정하는 것이 중요합니다.

6.2.2 은닉층의 활성화값 분포

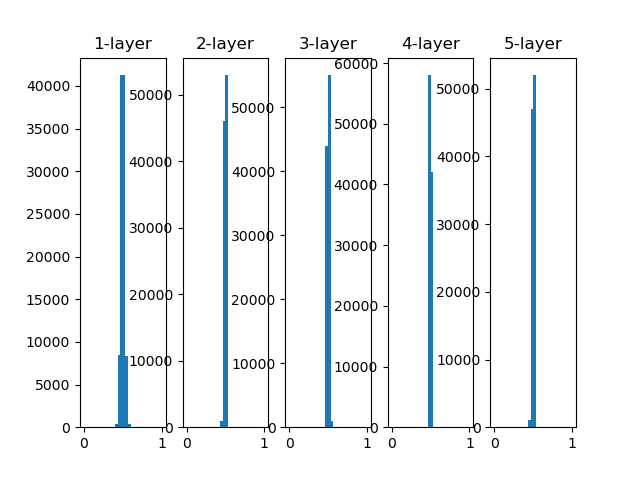
은닉층의 활성화값(활성화 함수의 출력 데이터)의 분포를 관찰하면 중요한 정보를 얻을 수 있습니다. 이번 절에서는 활성화 함수로 시그모이드 함수를 사용하는 5층 신경망에 무작위로 생성한 데이터를 흘리며 각 층의 활성화값 분포를 히스토그램으로 그려보고 가중치의 초깃값에 따라 은닉층의 활성화값들이 어떻게 변화하는지 보도록 하겠습니다. 다음은 표준편차가 1일 때의 활성화 값 분포입니다.

import numpy as np  
import matplotlib.pyplot as plt  
  
  
def sigmoid(x):  
 return 1 / (1 + np.exp(-x))  
  
  
x = np.random.randn(1000, 100)  
node\_num = 100  
hidden\_layer\_size = 5  
activations = {}  
  
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  
  
  
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()



히스토그램을 보면 0, 1에 활성화 값이 모여 있는 것을 볼 수 있습니다. 시그모이드 함수의 출력이 0과 1에 치우칠 때 시그모이드 함수의 미분 값은 0에 가까워집니다. 이런 경우에 역전파의 기울기 값은 점차 작아지다 사라지게 됩니다. 이것이 기울기 소실(gradient descent)라 불리며 층이 깊은 신경망에서 심각한 문제를 일으킵니다. 이번엔 표준편차를 0.01로 바꿔 실험을 계속 해보겠습니다. 다음과 같이 코드를 바꾸는 것으로 간단히 해볼 수 있습니다.

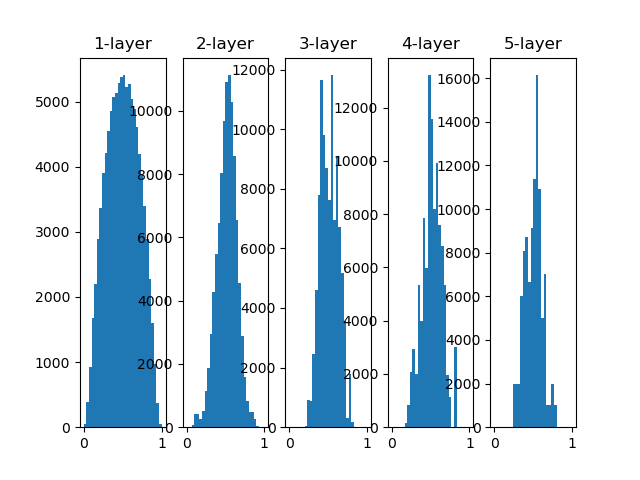
# w = np.random.randn(node\_num, node\_num) \* 1  
w = np.random.randn(node\_num, node\_num) \* 0.01



결과를 보면 0.5 부근에 집중되어 아까 같은 기울기 소실 문제는 발생하지 않습니다. 하지만 활성화 값이 치우쳤다는 것은 신경망의 표현력 관점에서는 큰 문제가 있습니다. 다수의 뉴련이 거의 같은 값을 출력하고 있으니 뉴런을 여러 개 둔 의미가 없어진다는 얘기입니다. 다음은 Xavier 초깃값을 사용해보겠습니다. Xavier 초깃값은 일반적인 딥러닝 프레임워크들이 표준적으로 이요하고 있습니다. Xavier의 논문에 따르면 노드의 개수가 n개 일 때 인 분포를 사용하면 된다고 합니다.

node\_num = 100

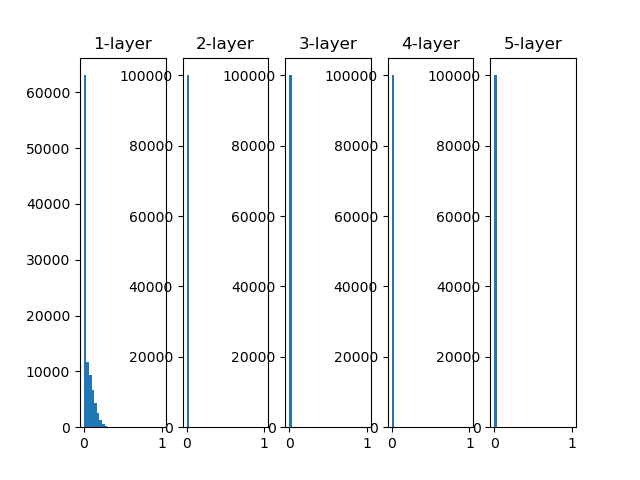
w = np.random.randn(node\_num, node\_num) / np.sqrt(node\_num)



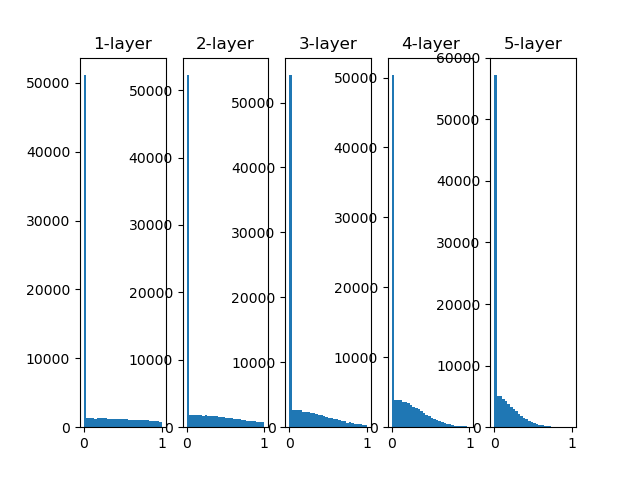
층이 깊어지면서 형태가 조금 일그러지지만, 앞선 방식보다는 확실히 넓게 분포되어 있습니다. 이런 일그러짐은 원점에서 대칭인 tanh 함수를 사용하면 개선된다고 합니다.

6.2.3 ReLU를 사용할 때의 가중치 초깃값

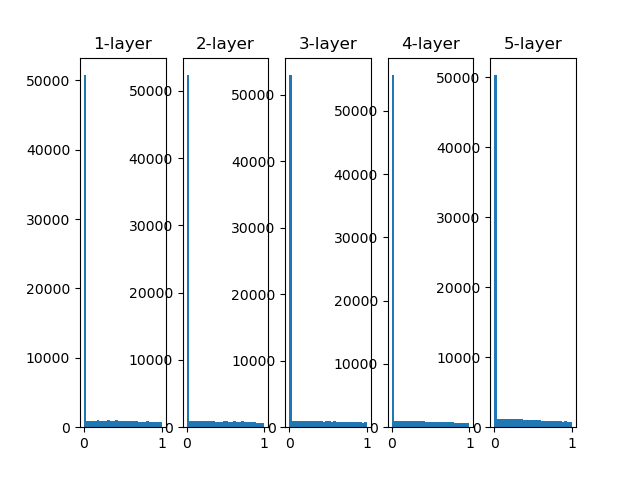
Xavier 초깃값은 활성화 함수가 선형인 것을 전제로 이끈 결과라고 합니다. sigmoid, tanh 함수는 좌우대칭이라 중앙 부근을 선형인 함수로 볼 수 있어 Xavier 초깃값이 적당합니다. 하지만 ReLU는 그렇지 않기 때문에 He 초깃값을 사용하라고 합니다. He 초깃값은 앞 계층의 노드가 n개일 때, 표준편차 인 정규분포를 사용합니다. 그러면 ReLU를 이용한 경우를 살펴보도록 하겠습니다. 실험은 표준편차가 0.01인 정규분포, Xavier 초깃값, He 초깃값일 때의 실험 결과를 차례로 보여줍니다.



표준편차가 0.01인 정규분포를 가중치 초깃값으로 사용한 경우



Xavier 초깃값을 사용한 경우



He 초깃값을 사용한 경우

결과를 보면 std=0.01일 때의 활성화 값들은 매우 작은 값들이고 이 뜻은 역전파 때 가중치의 기울기 역시 작아진다는 뜻입니다. 거의 학습이 이뤄지지 않을 것입니다.

Xavier 초깃값은 층이 깊어지면서 치우침이 점차 커집니다. 학습할 때 ‘기울기 소실’문제를 일으킵니다.

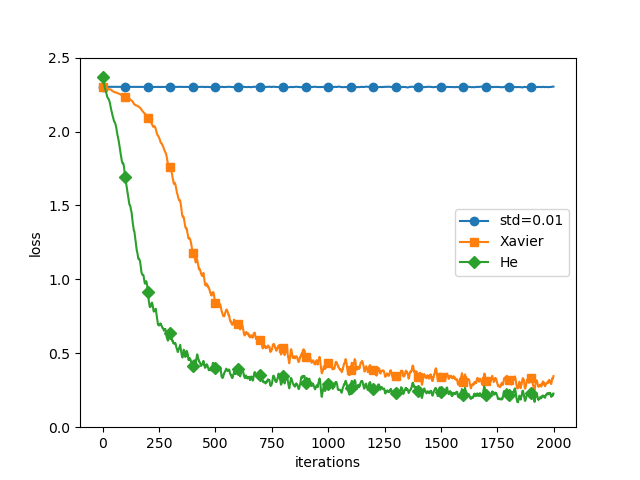
He 초깃값은 모든 층에서 균일하게 분포되고, 층이 깊어져도 분포가 유지되기에 학습이 잘 이뤄질 것입니다.

이 실험 결과를 바탕으로 이후에 활성화 함수로 ReLU를 사용할 때는 He 초깃값, sigmoid나 tanh 등의 S자 곡선일 때는 Xavier 초깃값을 사용하도록 하겠습니다.

6.2.4 MNIST 데이터셋으로 본 가중치 초깃값 비교

책의 예제 소스인 weight\_init\_compare.py를 통해 실제 데이터를 가지고 가중치의 초깃값을 설정하는 방법이 신경망 학습에 얼마나 영향을 주는지 알아보겠습니다.

# coding: utf-8  
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.util import smooth\_curve  
from common.multi\_layer\_net import MultiLayerNet  
from common.optimizer import SGD  
  
  
# 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==========  
weight\_init\_types = {'std=0.01': 0.01, 'Xavier': 'sigmoid', 'He': 'relu'}  
optimizer = SGD(lr=0.01)  
  
networks = {}  
train\_loss = {}  
for key, weight\_type in weight\_init\_types.items():  
 networks[key] = MultiLayerNet(input\_size=784, hidden\_size\_list=[100, 100, 100, 100],  
 output\_size=10, weight\_init\_std=weight\_type)  
 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 weight\_init\_types.keys():  
 grads = networks[key].gradient(x\_batch, t\_batch)  
 optimizer.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 weight\_init\_types.keys():  
 loss = networks[key].loss(x\_batch, t\_batch)  
 print(key + ":" + str(loss))  
  
  
# 3. 그래프 그리기==========  
markers = {'std=0.01': 'o', 'Xavier': 's', 'He': 'D'}  
x = np.arange(max\_iterations)  
for key in weight\_init\_types.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, 2.5)  
plt.legend()  
plt.show()



이 신경망은 활성화 함수로 ReLU를 이용하고 층별 뉴런 수가 100개인 5층 신경망입니다. 이전에 실험했던 결과와 비슷한 양상으로 std=0.01 일 때는 학습이 이뤄지지 않고 있고, Xavier 초깃값 보다는 He 초깃값이 학습 진도가 더 빠른 것을 볼 수 있습니다.

지금까지 본 것처럼 가중치의 초깃값은 신경망 학습에 중요한 포인트입니다. 가중치의 초깃값에 따라 신경망 학습의 성패가 갈리는 경우도 많기 때문에 주의하며 초깃값을 설정하도록 합시다.

6.3 배치 정규화

앞선 절에서 가중치 초깃값을 적절히 설정하면 각 층의 활성화값 분포가 적당히 퍼지면서 학습이 원활하게 이뤄진 것을 볼 수 있었습니다. 그렇다면 강제로 활성화값을 적당히 퍼뜨리면 어떻게 될까요? 이런 아이디어에서 출발한 것이 배치 정규화(batch normalization)입니다.

6.3.1 배치 정규화 알고리즘

배치 정규화는 2015년에 제안된 방법으로 나온지 얼마 안 된 기법임에도 여러 연구자와 기술자들이 즐겨 사용하며 뛰어난 결과를 달성하고 있습니다. 배치 정규화가 주목받는 이유는 다음과 같습니다.

1. 학습을 빨리 진행할 수 있다(학습속도 개선).
2. 초깃값에 크게 의존하지 않는다(골치 아픈 초깃값 선택 문제 해결).
3. 오버피팅을 억제한다(드롭아웃 등의 필요성 감소).

데이터의 분포를 정규화 하는 ‘배치 정규화’는 다음과 같이 신경망에 삽입됩니다.

Affine

Batch

Norm

ReLU

Affine

Batch

Norm

ReLU

데이터

배치 정규화는 이름과 같이 학습 시 미니배치를 단위로 정규화합니다. 구체적으로는 데이터 분포가 평균이 0, 분산이 1이 되도록 정규화 합니다. 수식은 다음과 같습니다.

여기에는 미니배치 B={x1, x2, …, xm}이라는 m개의 입력 데이터의 집합에 대해 평균 와 분산 을 구합니다. 그리고 입력 데이터를 평균이 0, 분산이 1이 되게 정규화 합니다. 기호는 작은 값으로, 0으로 나누는 사태를 예방하는 역할을 합니다.

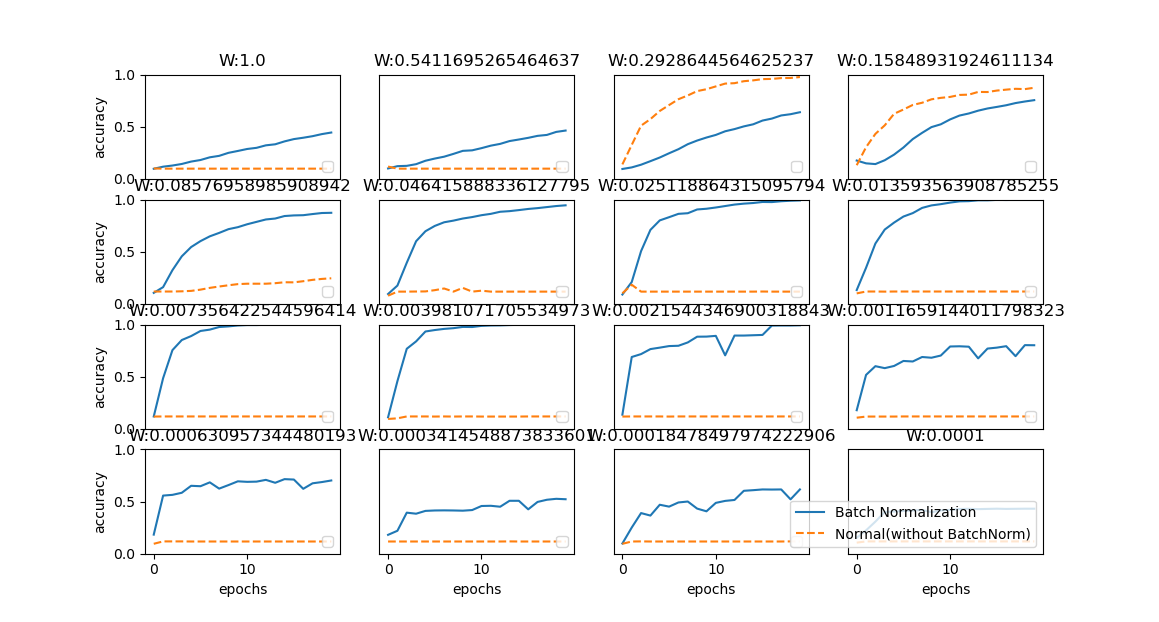
미니배치 입력 데이터를 평균 0, 분산 1인 데이터로 변환하는 간단한 작업을 하는 식으로, 이 처리를 활성화 함수의 앞 (혹은 뒤)에 삽입함으로써 데이터 분포가 덜 치우치게 할 수 있습니다.

또한, 배치 정규화 계층마다 이 정규화 된 데이터에 고유한 확대와 이동 변환을 수행합니다. 수식은 다음과 같습니다.

이 식에서 가 확대를, 는 이동을 담당합니다. 두 값은 처음에는 , 부터 시작하고 학습을 거치면서 적합한 값으로 조정해갑니다.

6.3.2 배치 정규화의 효과

배치 정규화를 사용한 실험을 해보도록 합시다. 책에서 제공한 소스 batch\_norm\_test.py를 사용하겠습니다.



거의 모든 경우에서 학습 진도가 빠른 것으로 나타납니다.

6.4 바른 학습을 위해

기계학습에서는 오버피팅이 문제가 되는 일이 많습니다. 오버피팅이란 신경망이 훈련 데이터에만 지나치게 적응되어서 그 외의 데이터에는 제대로 대응하지 못하는 상태를 말합니다. 아직 보지 못한 데이터가 주어져도 바르게 식별할 수 있어야 하는 신경망 모델에는 바람직하지 못한 상태입니다. 복잡하고 표현력이 높은 모델을 만들 수는 있지만, 범용성을 위해 오버피팅을 억제하는 기술 또한 중요합니다.

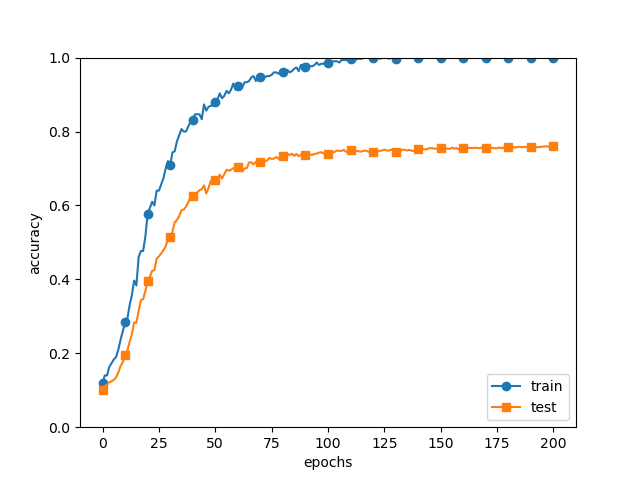
6.4.1 오버피팅

오버피팅은 주로 두가지 경우에 일어납니다.

1. 매개변수가 많고 표현력이 높은 모델
2. 훈련 데이터가 적음

이번 절에서는 일부러 오버피팅을 일으켜보도록 하겠습니다. 본래 60,000개인 MNIST 데이터셋의 훈련 데이터 중 300개만 사용하고, 7층 네트워크를 사용하여 복잡성을 높이고 각 층의 뉴런은 100개, 활성화 함수는 ReLU를 사용하도록 하겠습니다.

# coding: utf-8  
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=[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  
  
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', 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()



그래프를 보면 훈련데이터에서는 100 에폭을 지날 무렵부터 정확도가 100%이지만, 시험 데이터에 대해서는 그렇지 않습니다. 이렇게 큰 차이가 발생하는 이유는 모델이 훈련 데이터에 오버피팅했기 때문입니다.

6.4.2 가중치 감소

오버피팅을 억제하기 위해 많이 이용해온 방법 중에는 가중치 감소(weight decay)라는 것이 있습니다. 오버피팅은 가중치 매개변수의 값이 커서 발생하는 경우가 많기 때문에 큰 가중치에 대해서는 큰 페널티를 부과하여 오버피팅을 억제하는 방법입니다.

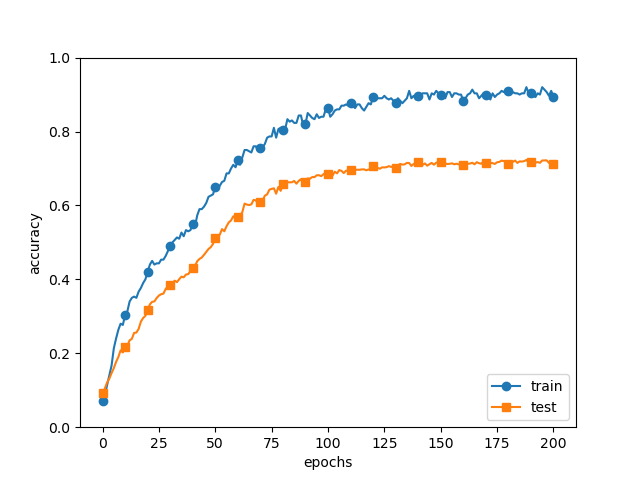
신경망 학습의 목적은 손실 함수의 값을 줄이는 것입니다. 이때, 가중치의 제곱 노름(norm, L2 norm)을 손실 함수에 더합니다. 그러면 가중치가 커지는 것을 억제할 수 있습니다. L2 노름은 각 원소의 제곱들을 더한 것입니다. L2 노름 외에 L1, L∞ 노름도 있습니다. L1 노름은 절댓값의 합, L∞ 노름은 Max 노름이라고도 하며, 각 원소의 절댓값 중 가장 큰 것에 해당합니다.

가중치를 W라 하면 L2 노름에 따른 가중치 감소는 이 되고 이 를 손실 함수에 더합니다. 는 정규화의 세기를 조정하는 하이퍼파라미터입니다.를 크게 설정할수록 큰 가중치에 대한 패널티가 커집니다. 은 의 미분 결과인 을 조정하는 역할의 상수입니다.

가중치 감소는 모든 가중치의 손실 함수에 을 더합니다. 따라서 가중치의 기울기를 구하는 계산에서는 오차역전파법에 따른 결과에 정규화 항을 미분한 을 더합니다.

그럼 실험을 해보도록 합시다.

# weight\_decay\_lambda = 0 # weight decay를 사용하지 않을 경우  
weight\_decay\_lambda = 0.1



6.4.3 드롭아웃

손실 함수에 가중치의 L2 노름을 더한 가중치 감소는 구현이 편하고 어느 정도 지나친 학습을 억제할 수 있습니다. 그러나 신경망 모델이 복잡해지면 가중치 감소만으로 오버피팅에 대응하기 어려워집니다. 이럴 때는 드롭아웃(Dropout)이라는 기법을 사용합니다.

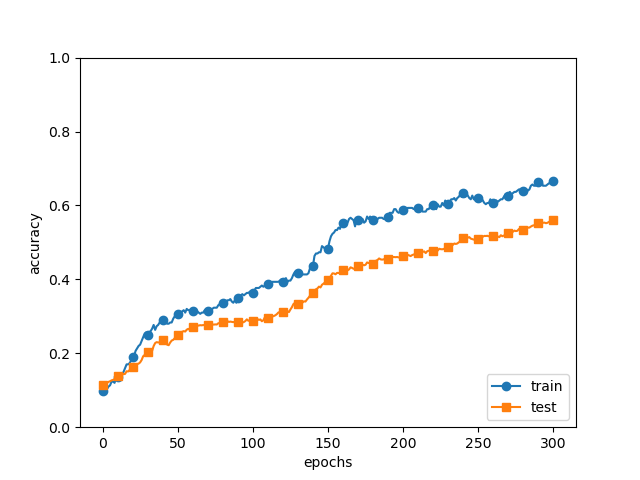
이 방법은 훈련 때에 뉴런을 임의로 삭제하며 학습합니다. 시험 때는 모든 뉴런에 신호를 전달합니다. 단, 시험 때는 각 뉴런의 출력에 훈련 때 삭제 안 한 비율을 곱하여 출력합니다.

간단하게 구현해 보도록 하겠습니다.

class Dropout:  
 def \_\_init\_\_(self, dropout\_ratio=0.5):  
 self.dropout\_ratio = dropout\_ratio  
 self.mask = None  
  
 def forward(self, x, train\_flag=True):  
 if train\_flag:  
 self.mask = np.random.randn(\*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

훈련 시에는 self.mask에 삭제할 뉴런을 False로 표시하여 역전파 때도 신호를 그대로 통과시킵니다. 드롭아웃 효과를 MNIST 데이터셋으로 확인해보겠습니다. 교재에서 구현을 간소화하기 위해 만든 Trainer 클래스를 사용했습니다.

import os, sys  
sys.path.append(os.pardir)  
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)  
  
x\_train = x\_train[:300]  
t\_train = t\_train[:300]  
  
use\_dropout = True  
dropout\_ratio = 0.2  
  
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  
  
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()



훈련 데이터와 시험 데이터에 대한 정확도 차이가 줄었습니다.

기계학습에서는 앙상블 학습(ensemble learning)을 애용합니다. 앙상블 학습은 개별적으로 학습시킨 여러 모델의 출력을 평균 내어 추론하는 방식입니다. 앙상블 학습은 드롭아웃과 밀접하다고 할 수 있는데, 이는 드롭아웃을 통해 뉴런을 삭제하는 행위를 매번 다른 모델을 학습시키는 것으로 해석할 수 있기 때문입니다. 또한 추론 때는 뉴런의 출력에 삭제한 비율을 곱함으로써 앙상블 학습에서 여러 모델의 평균을 내는 것과 같은 효과를 얻는 것입니다.

6.5 적절한 하이퍼파라미터 값 찾기

신경망에는 각 층의 뉴런 수, 배치 크기, 매개변수 갱신 시의 학습률과 가중치 등 여러가지 하이퍼파라미터가 존재합니다. 이런 하이퍼파라미터의 값을 적절히 설정하지 않으면 성능이 크게 떨어지는 만큼 중요하지만 그 값을 결정하기까지는 많은 시행착오를 겪습니다. 이번 절에서는 이러한 하이퍼파라미터 값을 최대한 효율적으로 탐색하는 방법에 대해 알아봅시다.

6.5.1 검증 데이터

지금까지는 데이터셋을 훈련 데이터와 시험 데이터라는 두 가지로 분리해 이용했습니다. 앞으로 하이퍼파라미터를 다양한 값으로 설정하고 검증할 텐데, 이 때는 시험 데이터를 사용해선 안 됩니다.

그 이유는 시험 데이터를 사용하여 하이퍼파라미터를 조정하면 하이퍼파라미터가 시험 데이터에 오버피팅 되어 버립니다. 그래서 하이퍼파라미터를 전용 확인 데이터가 필요한데, 이를 일반적으로 검증 데이터(validation data)라고 부릅니다.

MNIST 같은 데이터셋의 경우에는 훈련, 시험 데이터로만 나눠져 있기 때문에 사용자가 직접 검증 데이터를 나눠야 할 필요가 있습니다. 코드로는 다음과 같습니다.

(x\_train, t\_train), (x\_test, t\_test) = load\_mnist(normalize=True)  
  
x\_train = x\_train[:500]  
t\_train = t\_train[:500]  
  
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 shuffle\_dataset(x, t):  
permutation = np.random.permutation(x.shape[0])  
 x = x[permutation,:] if x.ndim == 2 else x[permutation,:,:,:]  
 t = t[permutation]  
  
 return x, t

이제 검증 데이터를 사용하여 하이퍼파라미터를 최적화하는 기법을 살펴봅시다.

6.5.2 하이퍼파라미터 최적화

하이퍼파라미터 최적화의 핵심은 ‘최적 값’이 존재하는 범위를 조금씩 줄여간다는 것입니다. 우선 대략적인 범위를 정하고 그 범위에서 무작위로 하이퍼파라미터 값을 골라낸 후 그 값으로 정확도를 평가합니다.

그 범위는 대략적으로 지정하는 것이 효과적이고, 0.001에서 1,000 사이와 같이 ‘10의 거듭제곱’ 단위로 범위를 지정합니다. 이를 ‘로그 스케일(log scale)로 지정한다’ 합니다.

하이퍼파라미터를 최적화할 때는 딥러닝 학습에 시간이 오래 걸리므로 나쁠 듯한 값은 일찍 포기하는 것이 좋습니다. 그래서 에폭을 작게 하여, 1회 평가에 걸리는 시간을 단축시킵니다. 정리하자면 다음과 같습니다.

* 0단계

하이퍼파라미터 값의 범위를 설정합니다.

* 1단계

설정된 범위에서 하이퍼파라미터의 값을 무작위로 추출합니다.

* 2단계

1단계에서 샘플링한 하이퍼파라미터 값을 사용하여 학습하고, 검증 데이터로 정확도를 평가합니다(단, 에폭은 작게 설정합니다).

* 3단계

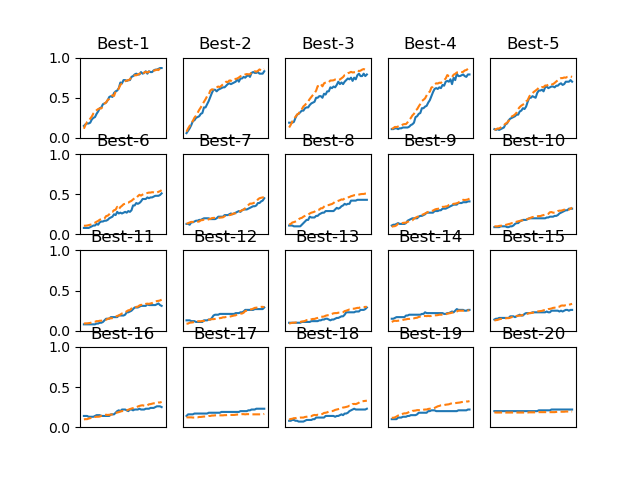
1단계와 2단계를 특정 횟수(100회 등) 반복하며, 그 정확도의 결과를 보고 하이퍼파라미터의 범위를 좁힙니다.

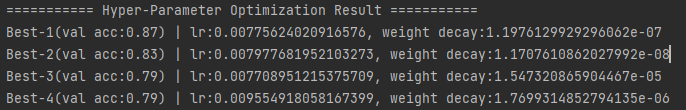
여기에서 설명한 최적화 방법은 실용적인 방법으로 수행자의 ‘지혜’와 ‘직감’에 의존한다는 느낌이 듭니다. 더 세련된 기법을 원한다면 베이즈 최적화(Bayesian optimization)을 공부해보는 것이 좋다고 합니다. 베이즈 최적화는 베이즈 정리(Bayes’ theorem)를 중심으로 한 수학 이론을 바탕으로 엄밀하고 효율적으로 최적화를 진행합니다.

6.5.3 하이퍼파라미터 최적화 구현하기

책에서 제공한 hyperparameter\_optimization.py 코드를 보도록 하겠습니다.

# coding: utf-8  
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 import MultiLayerNet  
from common.util import shuffle\_dataset  
from common.trainer import Trainer  
  
(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, epocs=50):  
 network = MultiLayerNet(input\_size=784, hidden\_size\_list=[100, 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=epocs, 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  
  
# 그래프 그리기========================================================  
print("=========== Hyper-Parameter Optimization Result ===========")  
graph\_draw\_num = 20  
col\_num = 5  
row\_num = int(np.ceil(graph\_draw\_num / col\_num))  
i = 0  
  
for key, val\_acc\_list in sorted(results\_val.items(), key=lambda x:x[1][-1], reverse=True):  
 print("Best-" + str(i+1) + "(val acc:" + str(val\_acc\_list[-1]) + ") | " + key)  
  
 plt.subplot(row\_num, col\_num, i+1)  
 plt.title("Best-" + str(i+1))  
 plt.ylim(0.0, 1.0)  
 if i % 5: plt.yticks([])  
 plt.xticks([])  
 x = np.arange(len(val\_acc\_list))  
 plt.plot(x, val\_acc\_list)  
 plt.plot(x, results\_train[key], "--")  
 i += 1  
  
 if i >= graph\_draw\_num:  
 break  
  
plt.show()





결과를 보며 학습이 잘 진행될 때의 학습률과 가중치 감소 계수를 관측하고 범위를 좁혀갑니다. 이렇게 좁혀가다가 최종 하이퍼파라미터 값을 하나 선택하면 됩니다.