

CUAI DLS스터디 2팀

2022.03.08

발표자 : 배준학, 임유민

스터디원 소개 및 만남 인증

사진을 깜박했습니다...!!

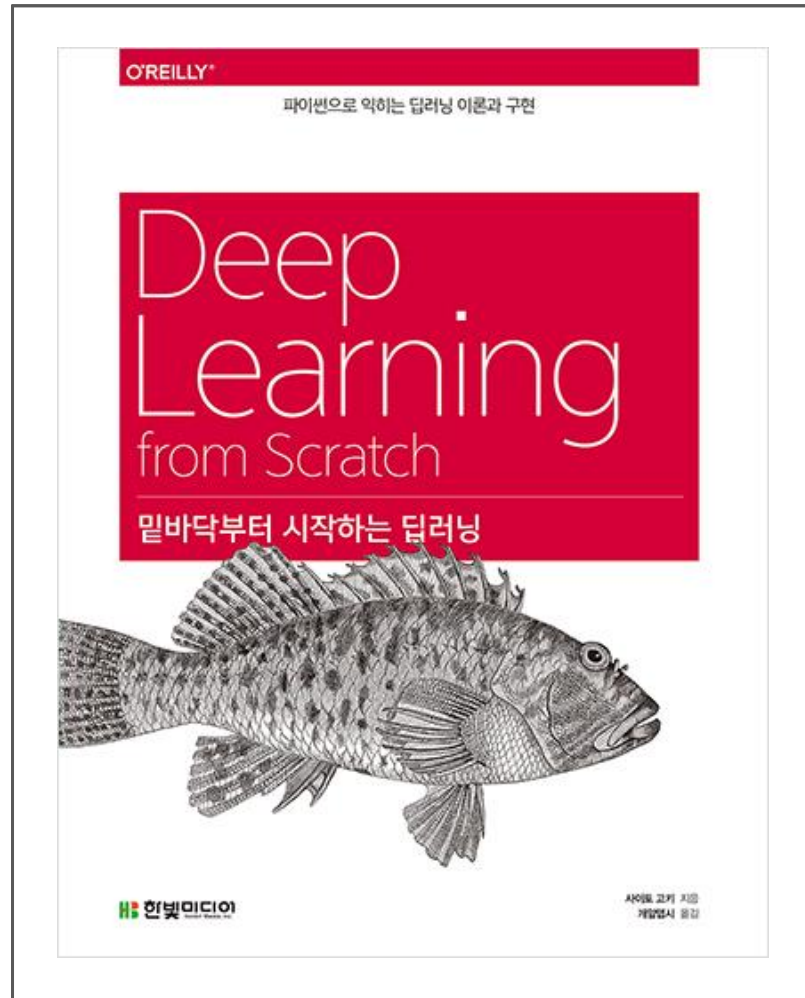
스터디원 1 : 박수빈

스터디원 2 : 배준학

스터디원 3 : 임유민

스터디원 4 : 최형용

스터디 계획



- ✓ 일시 : 매주 일요일 18시
- ✓ 방법 : 비대면 미팅(discord), 각자 공부한 내용 토대로 질문 형성 및 토의 진행
- ✓ 진도 : 한 주에 한 장 공부하기
- ✓ 목표 : 이번 학기까지 딥러닝 1권 끝내기
- ✓ 챕터6: 학습 관련 기술들

1. 매개변수 갱신

신경망 학습의 목적 = 최적화

: 손실함수의 값을 최소화하는 매개변수를 찾는 것

최적화 기법의 종류

1.SGD

2.모멘텀

3.AdaGrad

4.Adam

1. 매개변수 갱신

1. SGD

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

- \mathbf{W} 는 갱신할 가중치 매개변수이다.
- L / \mathbf{W} 은 \mathbf{W} 에 대한 손실 함수의 기울기이다.
- (에타)는 학습률을 의미한다. 하이퍼파라미터이므로 0.01이나 0.001 같은 값을 미리 정해서 사용한다.
- 우변의 값으로 좌변의 값을 갱신한다는 뜻이다

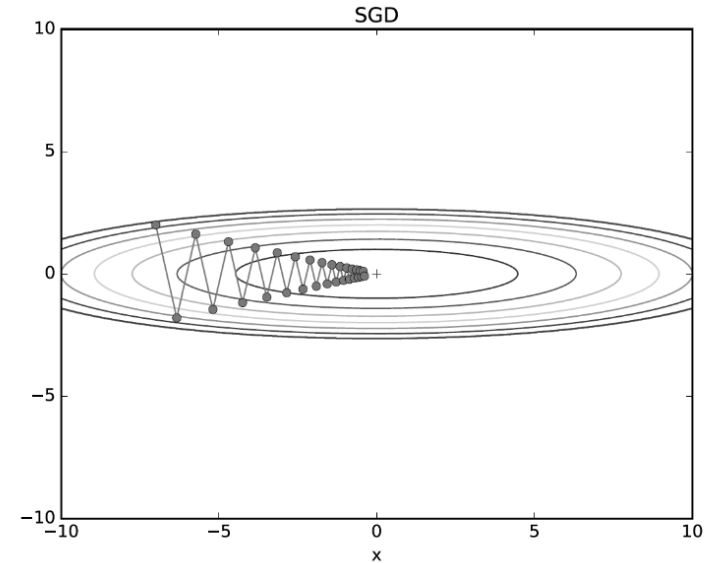
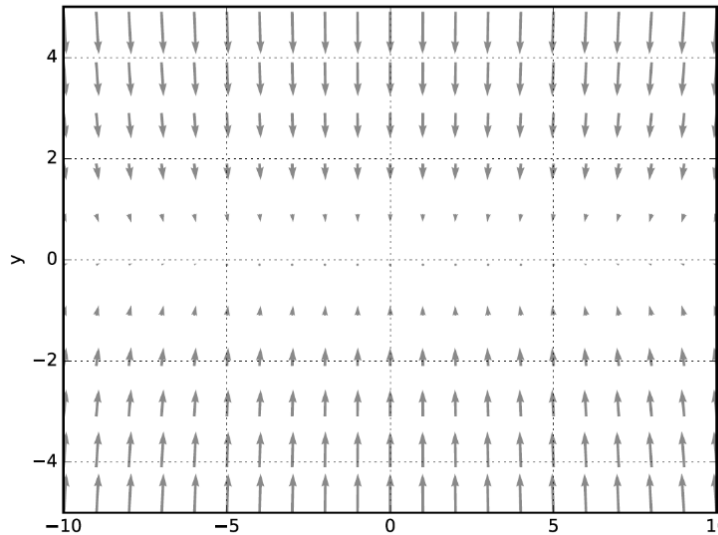
1. 매개변수 갱신

1.1 SGD 단점

SGD는 단순하고 구현도 쉽지만, 문제에 따라 비효율적일 때가 있다.

$f(x,y)$ 기울기 그래프

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



- $f(x,y)$ 가 최소가 되는 점은 $(x,y)=(0,0)$ 이지만 위의 그림에서 보여주는 기울기 대부분은 $(0,0)$ 방향을 가리키지 않음
- SGD가 지그재그로 탐색하게 되는 원인은 기울어진 방향이 본래의 최솟값과 다른 방향을 가리키기 때문이다.
- 무작정 기울어진 방향으로 진행하다 보니, 비효율적으로 탐색하게 됨

1. 매개변수 갱신

2. 모멘텀

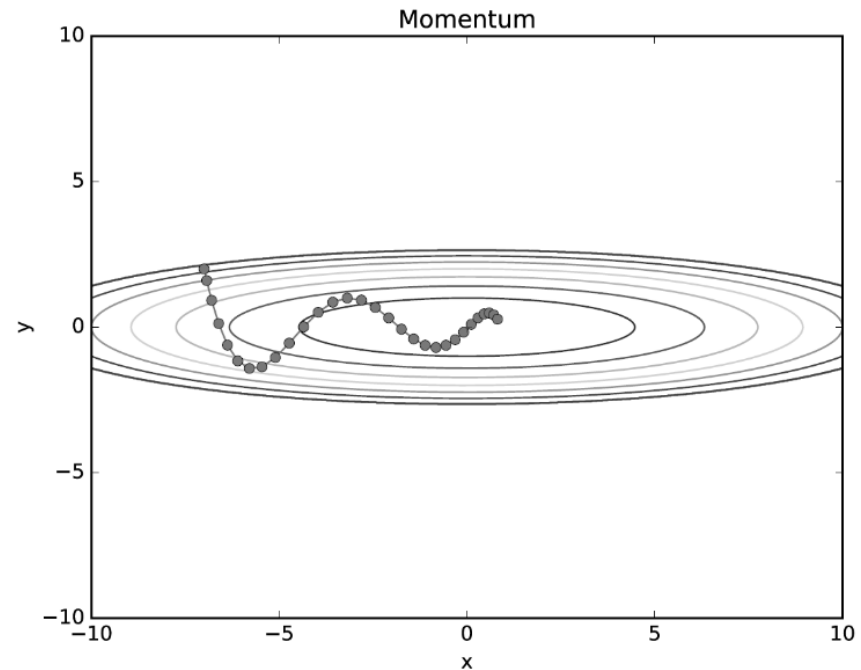
$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

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

\mathbf{v} 추가

- \mathbf{v} 는 물리에서 말하는 속도(velocity)에 해당한다.
- \mathbf{v} 항은 물체가 아무런 힘을 받지 않을 때 서서히 하강시키는 역할을 한다.
- 는 0.9 등으로 설정한다. (물리에서 지면 마찰이나 공기저항)
- 전체적으로는 SGD보다 x축 방향으로 빠르게 다가가 지그재그 움직임이 줄어든다.

모멘텀 적용



파이썬

1. 매개변수 갱신

3. AdaGrad

학습률 감소 기술: 학습을 진행하면서 학습률을 점차 줄여가는 방법 (처음엔 크게 나중에 작게)

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

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

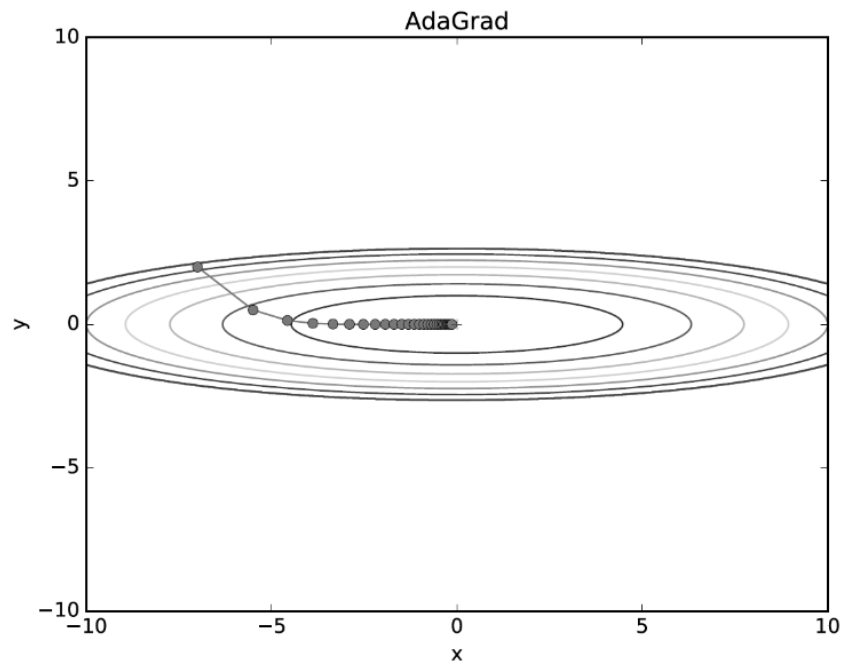
h 등장

- h는 식에서 보듯 기존 기울기 값을 제공하여 계속 더해준다.
- 매개변수를 갱신할 때 $1/h$ 을 곱해 학습률을 조정.
- 매개변수의 원소 중에서 많이 움직인(크게 갱신된) 원소는 학습률이 낮아지고, 학습이 진행될 수록 학습률이 낮아짐

1. 매개변수 갱신

3. AdaGrad

AdaGrad 적용



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

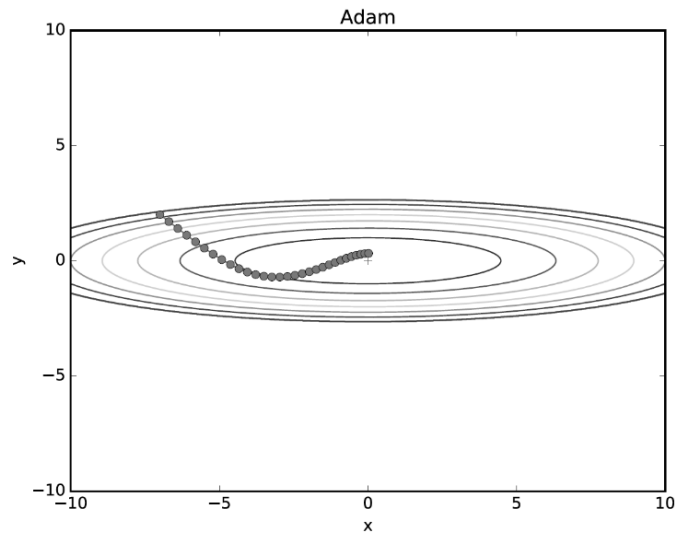
파이썬

1. 매개변수 갱신

4. Adam

위의 두 방법을 융합한 기법

Adam 적용



- 위 그림에서 보이듯 Adam 갱신 과정도 그릇 바닥을 구르듯 움직인다.
- 모멘텀과 비슷한 패턴인데, 모멘텀 때보다 공의 좌우 흔들림이 적다.
- 이는 학습의 갱신 강도를 적응적으로 조정해서 얻는 혜택이다.

1. 매개변수 갱신

6.1.6 어떤 갱신 방법을 이용해야 하나?

=> SGD를 포함한 네가지 방법 중 모든 문제에 뛰어난 기법은 없음.

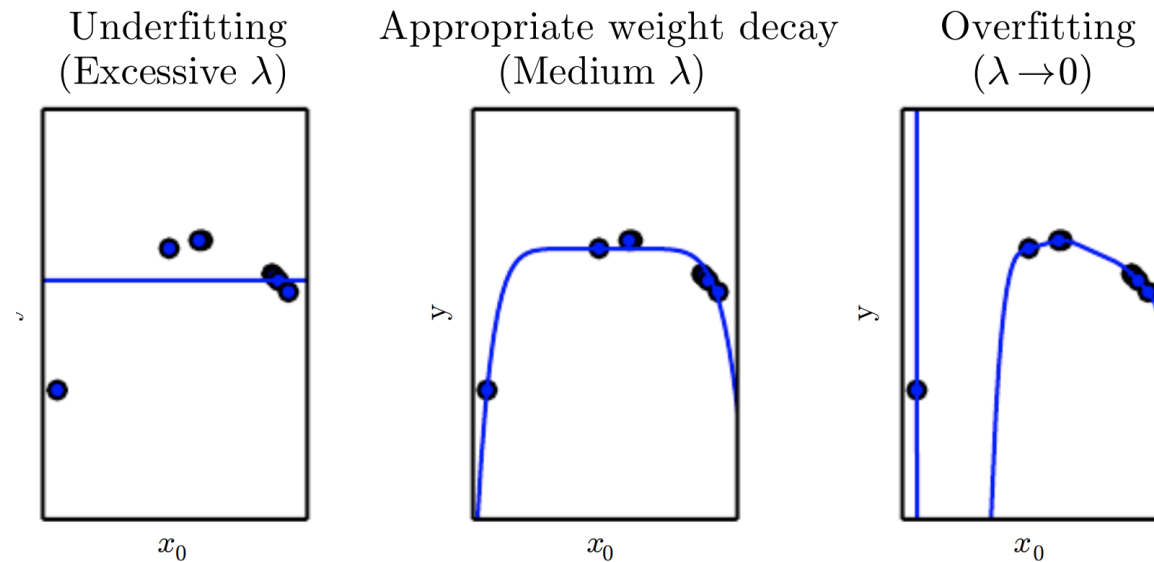
- 문제에 따라, 학습률 등의 파라미터 설정에 따라 달라짐.
- 각자 상황을 고려해 여러가지로 시도하는 것이 중요

2. 가중치의 초깃값

1) 초깃값을 0으로 하면?

“가중치 감소(Weight Decay) 기법”

: 가중치 매개변수의 값이 작아지도록 학습하는 방법 -> 오버피팅 방지

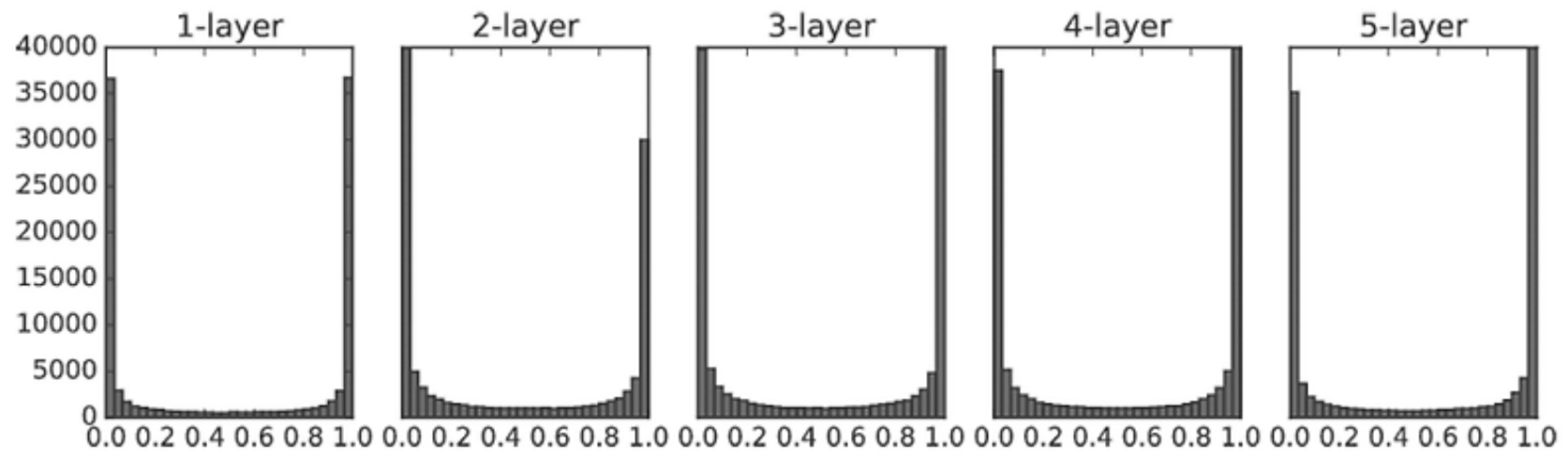


그렇다면, 가중치 초깃값을 0으로 하면 어떻게 될까? -> 가중치가 고르게 되어버림!!

2. 가중치의 초깃값

2) 은닉층의 활성화값 분포

(1) 가중치를 표준편차가 1인 정규분포로 초기화

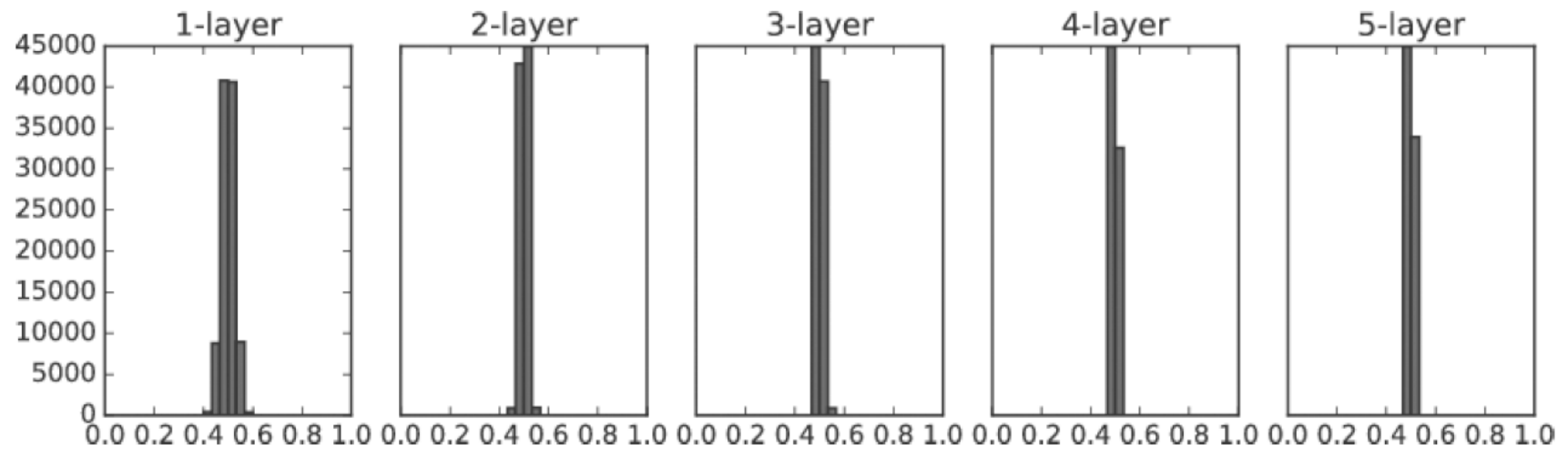


활성화 값들이 0과 1에 치우쳐 분포되어 있다 -> 기울기 소실(Gradient Vanishing)

2. 가중치의 초깃값

2) 은닉층의 활성화값 분포

(2) 가중치를 표준편차가 0.01인 정규분포로 초기화



0.5 부근에 분포가 집중된다 -> 다수의 뉴런이 거의 같은 값을 출력 -> 표현력의 제한

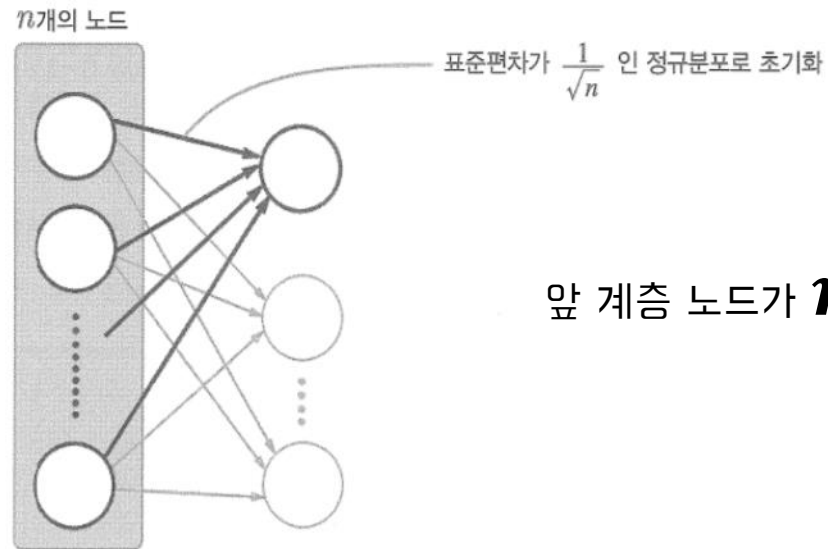
2. 가중치의 초기값

2) 은닉층의 활성화값 분포

(3) Xavier 초기값!

“ Xavier 초기값 ”

: 일반적인 딥러닝 프레임워크에서 표준적으로 이용되는 초기값

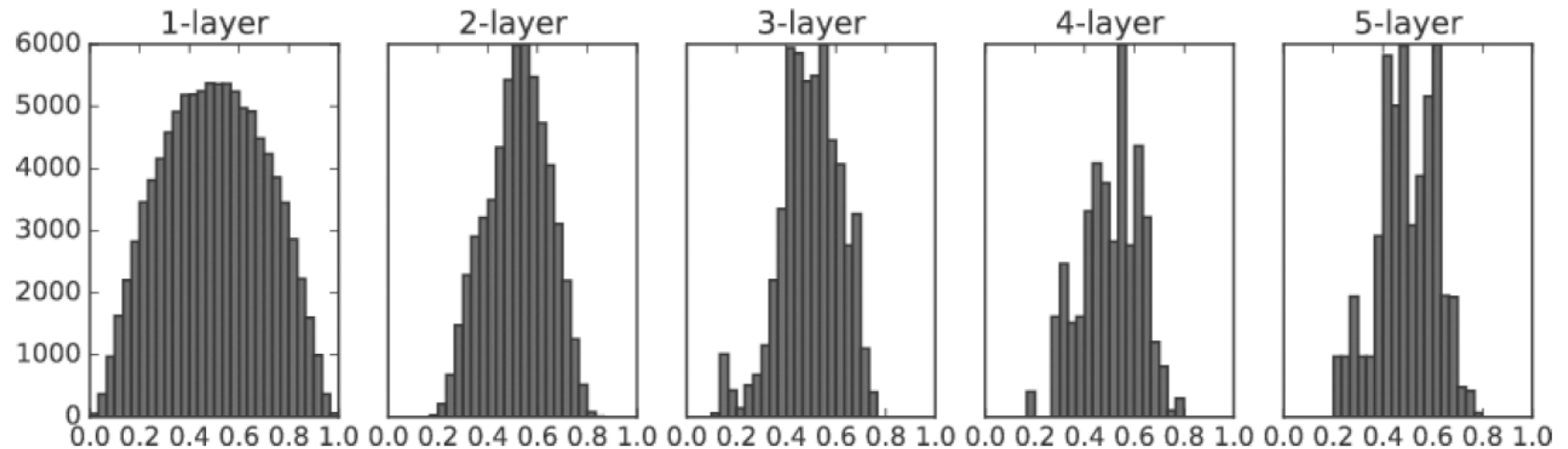


앞 계층 노드가 n 개일 때, 표준편차가 $\frac{1}{\sqrt{n}}$ 인 분포를 사용

2. 가중치의 초깃값

2) 은닉층의 활성화값 분포

(3) Xavier 초깃값!



-> 앞의 두 방법보다 넓게 분포됨!

2. 가중치의 초기값

3) ReLU를 사용할 때의 가중치 초기값

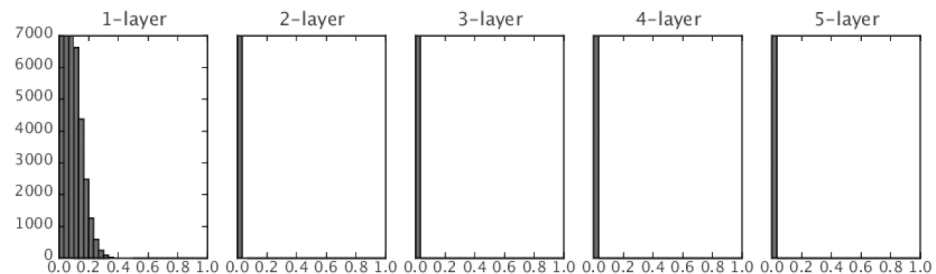
Xavier 초기값은 활성화 함수가 선형일 때 사용 -> sigmoid 함수와 tanh 함수에 적합

그렇다면 ReLU 함수는? -> “ He 초기값 “

: 앞 계층의 노드가 n 개일 때, 표준편차가 $\sqrt{\frac{2}{n}}$ 인 정규분포를 사용

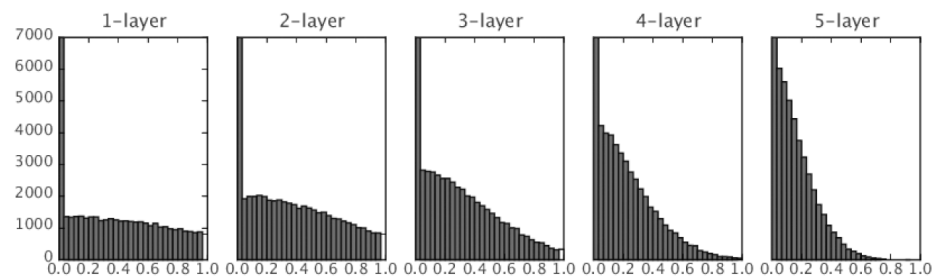
2. 가중치의 초깃값

3) ReLU를 사용할 때의 가중치 초깃값



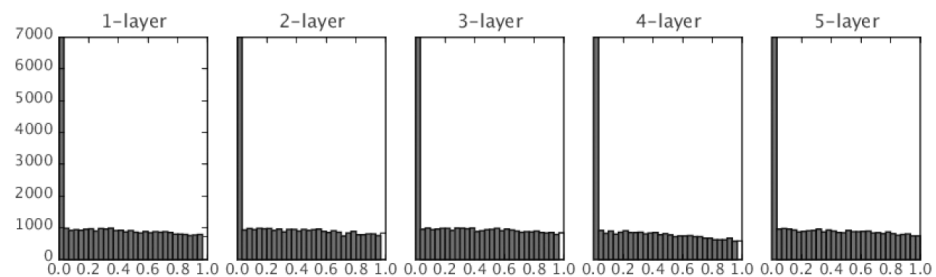
표준편차가 0.01인 정규분포를 가중치 초깃값으로 사용한 경우

1) 표준편차가 0.01인 정규분포



Xavier 초깃값을 사용한 경우

2) Xavier 초깃값



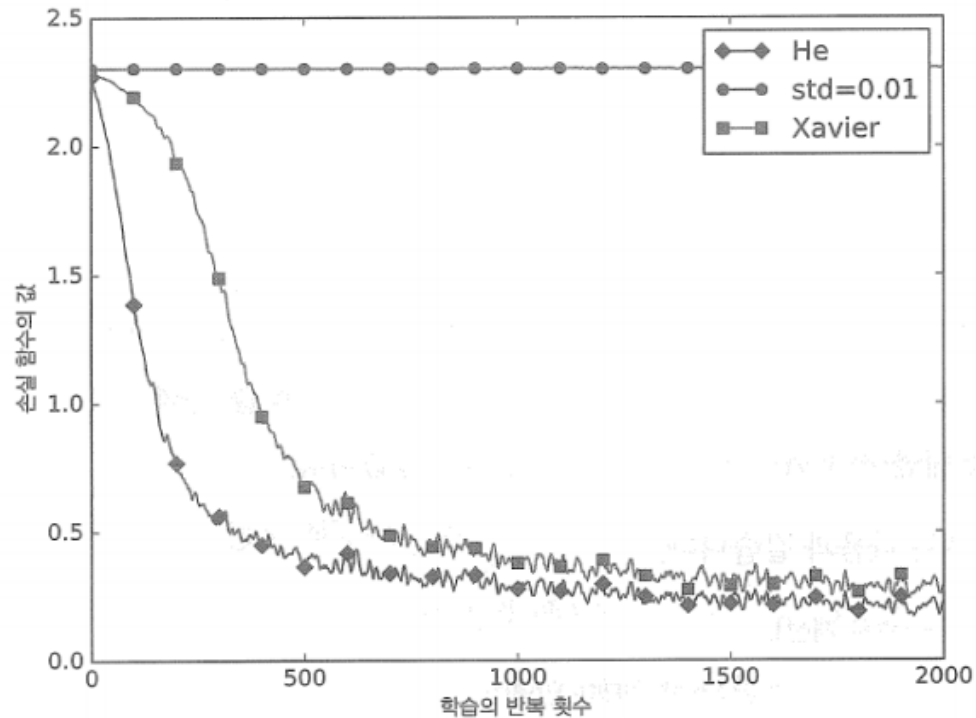
He 초깃값을 사용한 경우

3) He 초깃값

2. 가중치의 초깃값

4) MNIST 데이터셋으로 본 가중치 초깃값 비교

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



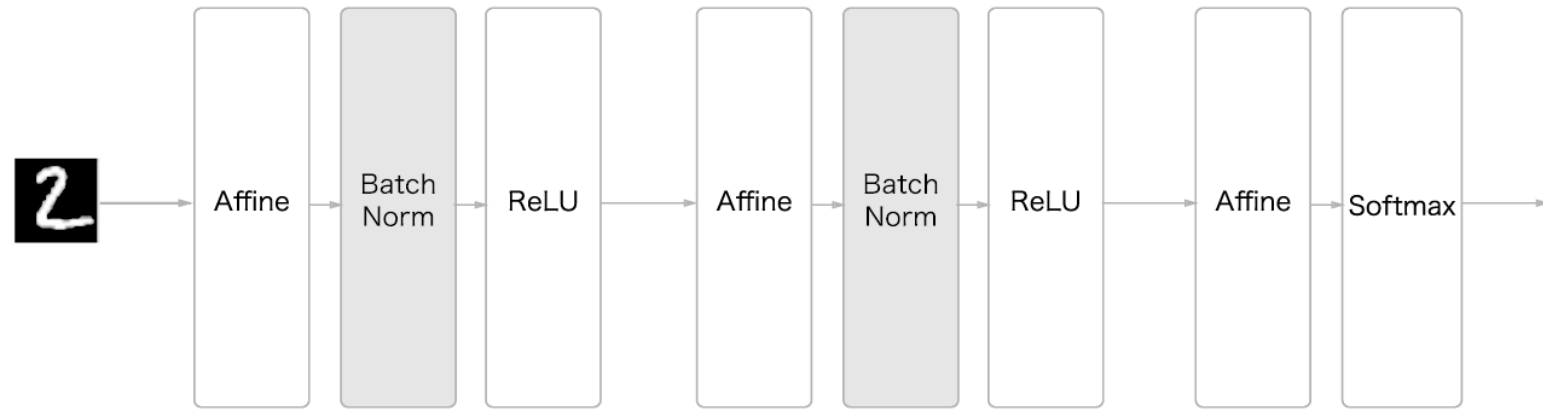
- std=0.01 : 학습이 전혀 이루어지지 않음
- He : 학습이 빠르게 진행됨
- Xavier : 학습이 진행됨

-> 초깃값은 신경망 학습의 성패를 결정한다!

3. 배치 정규화

1) 배치 정규화 알고리즘

: 각 층이 활성화를 적당히 퍼뜨리도록 '강제' 하는 것



(데이터 분포를 정규화 하는 '배치 정규화 계층'을 신경망 사이사이에 삽입한다.)

< 장점 >

- (1) 학습 속도가 빨라진다.
- (2) 초깃값에 크게 의존하지 않는다.
- (3) 오버피팅을 억제한다.

3. 배치 정규화

1) 배치 정규화 알고리즘

- 데이터 분포가 [평균이 0, 분산이 1]이 되도록 정규화

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

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

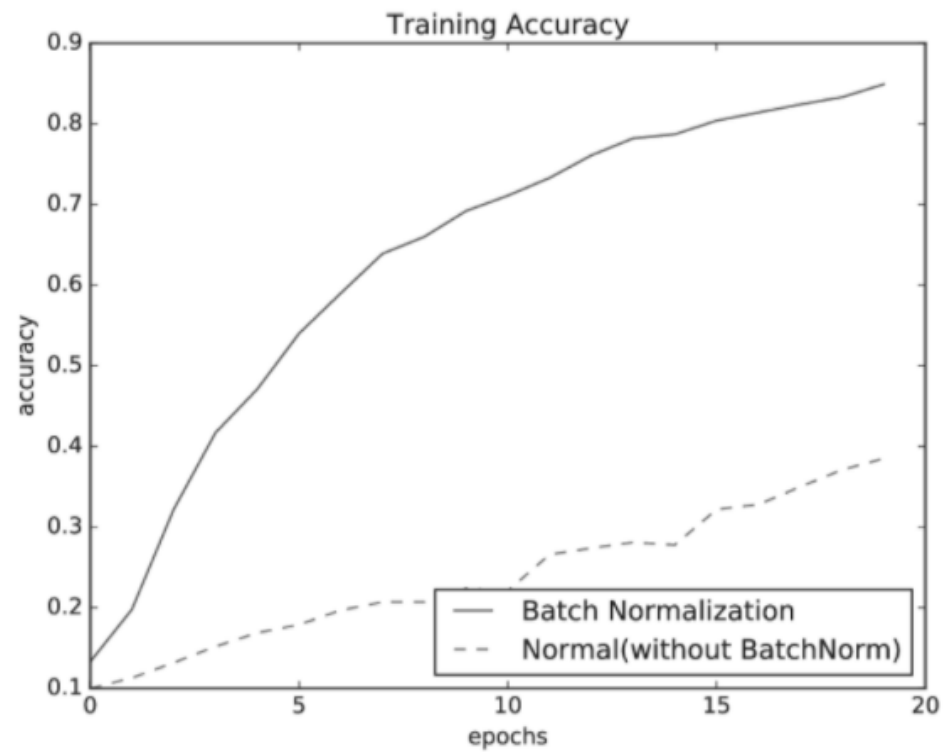
$$\hat{x} \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

- 정규화된 데이터에 고유한 확대(scale)와 이동(shift) 변환을 수행

$$y_i = \gamma \hat{x}_i + \beta$$

3. 배치 정규화

1) 배치 정규화의 효과



-> 배치 정규화를 했을 때 학습 속도가 훨씬 향상됨!

4. 바른 학습을 위해

기계 학습에서는 **오버피팅**이 문제가 되는 경우가 많다.

오버피팅: 신경망이 훈련 데이터에만 지나치게 적응되어 그 외의 데이터에는 제대로 대응하지 못하는 상태

=> 오버피팅을 억제하는 기술이 중요하다.

> 오버피팅

오버피팅이 일어나는 경우

- 매개변수가 많고 표현력이 높은 모델
- 훈련 데이터가 적음

>> 60000개의 훈련 데이터 중 300개만 사용. 7층 네트워크를 사용해 네트워크 복잡성을 높임.

>> 각 층의 뉴런은 100개 활성화 함수는 ReLU를 사용

4. 바른 학습을 위해

> 오버피팅

```
from dataset.mnist import load_mnist

(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]
```

```
import numpy as np
import matplotlib.pyplot as plt
from common.multi_layer_net import MultiLayerNet
from common.optimizer import SGD

# weight decay (가중치 감소) 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], 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
```

```
for i in range(1000000000):
```

```
    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
```

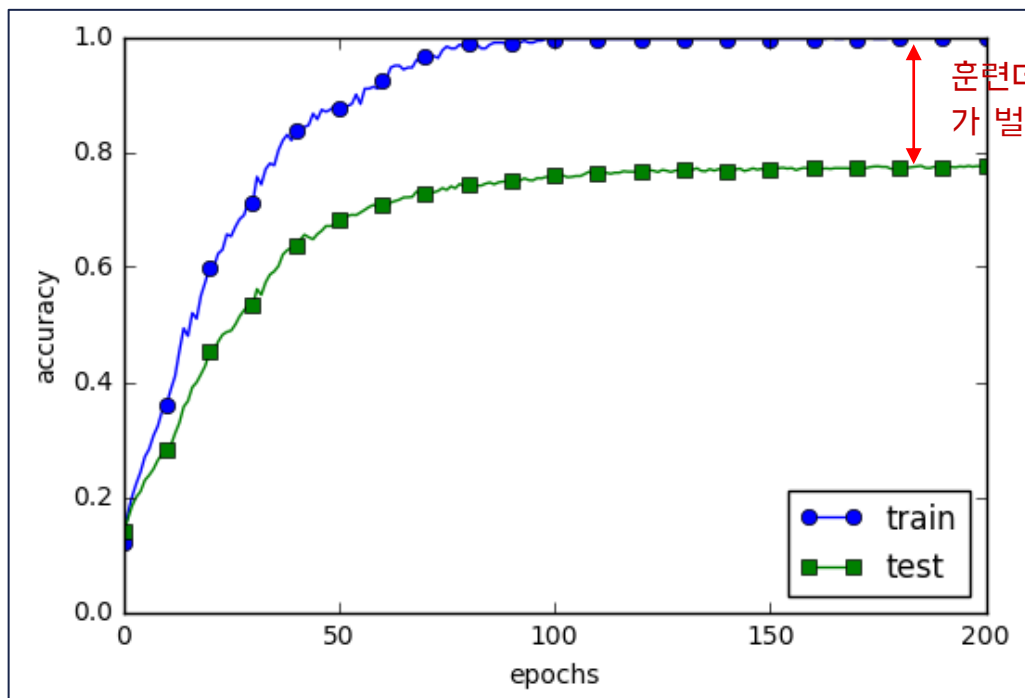
에폭 단위의 정확도를 저장

4. 바른 학습을 위해

> 오버피팅

```
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()
```

[그래프 시각화]



훈련데이터와 학습데이터에서의 정확도 차이가 벌어진다 -> 오버피팅 발생

4. 바른 학습을 위해

> 가중치 감소

- 학습 과정에서 큰 가중치에 대해서는 그에 상응하는 큰 패널티를 부과하여 오버피팅을 억제하는 방법
- 가중치를 W 라 하면 L2 법칙에 따른 가중치 감소는 $1/2 (W^2)$ 가 되고 이 값을 손실함수에 더함
- λ (람다)는 정규화의 세기를 조절하는 하이퍼파라미터. 이 값을 크게 설정할 수

> L2 법칙 가중치에 대한 패널티가 커짐

$W = (W_1, W_2, \dots, W_n)$ 의 이차 노름 (L2 노름)

$$\sqrt{W_1^2 + W_2^2 + \dots + W_n^2}$$

4. 바른 학습을 위해

> 가중치 감소

[가중치 감소(=0.1)를 적용한 결과]

```
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(100000000):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

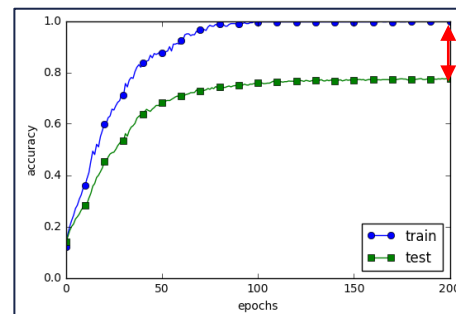
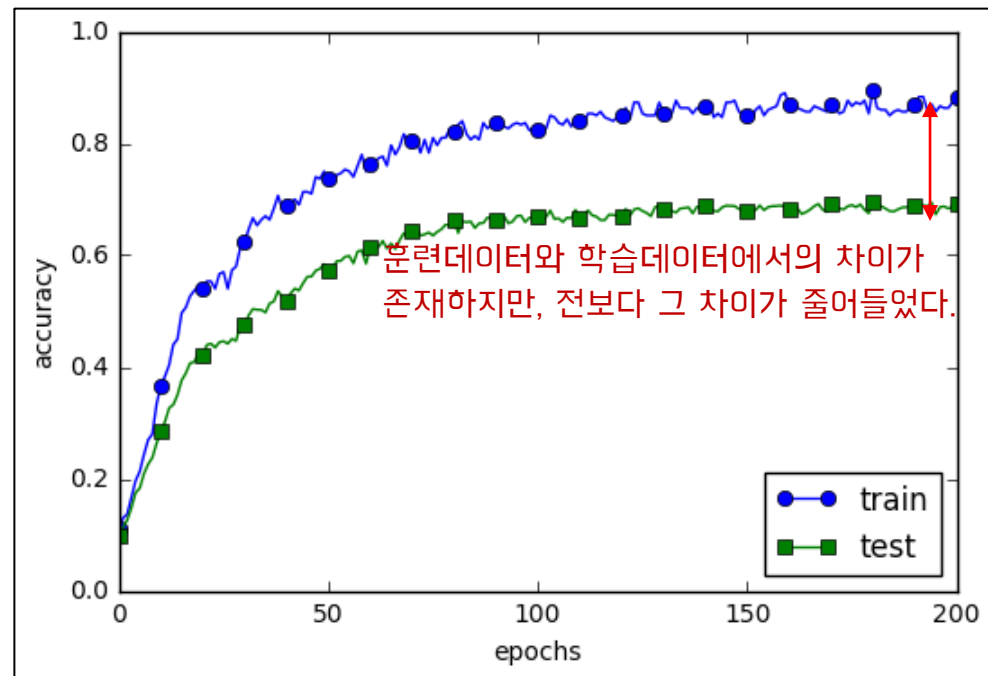
    grads = network.gradient(x_batch, t_batch)
    optimizer.update(network.params, grads)

    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()
```



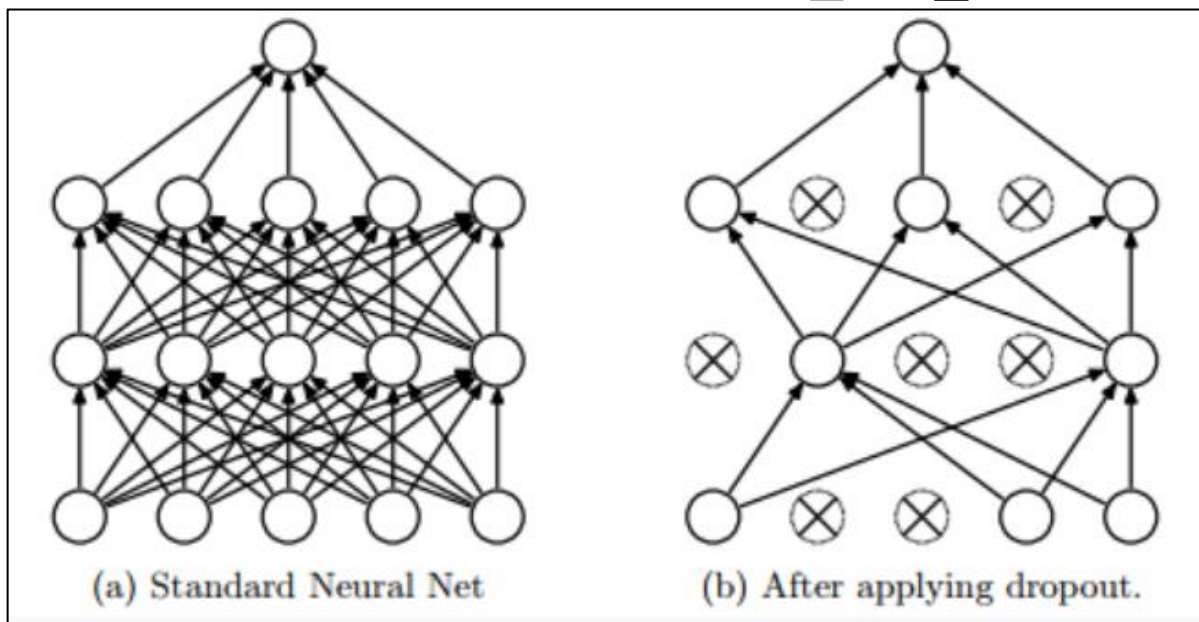
4. 바른 학습을 위해

> 드롭아웃

신경망 모델이 복잡해지면 **가중치 감소만으로는 대응하기 어려움**

드롭아웃 : 뉴런을 임의로 삭제하면서 학습하는 방법

- 훈련 때에는 데이터를 흘릴 때마다 삭제할 뉴런을 무작위로 선택.
- 시험 때에는 모든 뉴런에 신호를 전달. 각 뉴런의 출력에 훈련 때 삭제한 비율



4. 바른 학습을 위해

> 드롭아웃

드롭아웃 구현

```
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
```

- 훈련 시에는 순전파 때마다 self.mask에 삭제할 뉴런을 False로 표시
- 역전파 때의 동작은 ReLU와 같음.
- 순전파 때 통과시키지 않은 뉴런은 역전파 때도 신호를 차단.

```
# coding: utf-8
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)

# 오버피팅을 재현하기 위해 학습 데이터 수를 줄임
x_train = x_train[:300]
t_train = t_train[:300]

# 드롭아웃 사용 유무와 비율 설정 =====
use_dropout = True # 드롭아웃을 쓰지 않을 때는 False
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_ratio=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=False)
trainer.train()

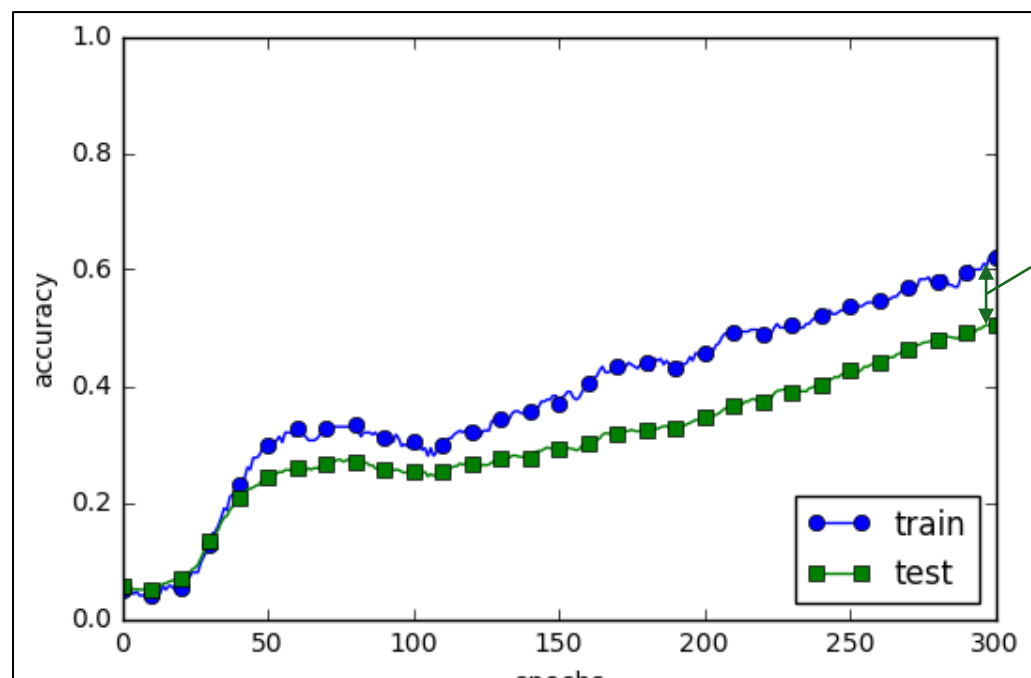
train_acc_list, test_acc_list = trainer.train_acc_list, trainer.test_acc_list

# 그래프 그리기=====
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()
```

4. 바른 학습을 위해

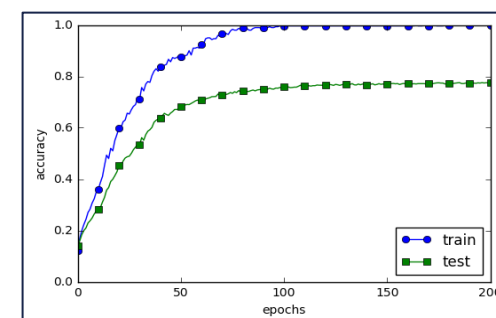
> 드롭아웃

[드롭아웃 적용한 결과]



훈련 데이터와 테스트 데이터의 정확도가 줄어들었다.

[드롭아웃 비적용]



드롭아웃을 이용하면 표현력을 높이면서 오버피팅을 억제가능


5. 적절한 하이퍼파라미터 값 찾기

5.1 검증데이터

하이퍼파라미터란?

머신러닝 모델의 구성 요소로, 모델 학습 과정에서 자동으로 학습되지 않고 외부에서 설정되는 파라미터를 의미

하이퍼파라미터 성능 평가

시험 데이터 사용 **X**  하이퍼파라미터 전용 확인 데이터가 필요

이를 검증 데이터(validation data)라고 부른다.

Why?

시험 데이터를 사용하여 조정하면 하이퍼파라미터 값이 시험 데이터에 오버피팅되기 때문이다.

5. 적절한 하이퍼파라미터 값 찾기

5.2 하이퍼파라미터 최적화

핵심은 하이퍼파라미터의 '최적 값'이 존재하는 범위를 조금씩 줄여간다는 것이다.

우선 대략적인 범위를 설정하고 그 범위에서 무작위로 하이퍼파라미터 값을 골라낸 후, 그 값으로 정확도를 평가한다.

- **0단계**

하이퍼파라미터 값의 범위를 설정한다.

- **1단계**

설정된 범위에서 하이퍼파라미터의 값을 무작위로 추출한다.

- **2단계**

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

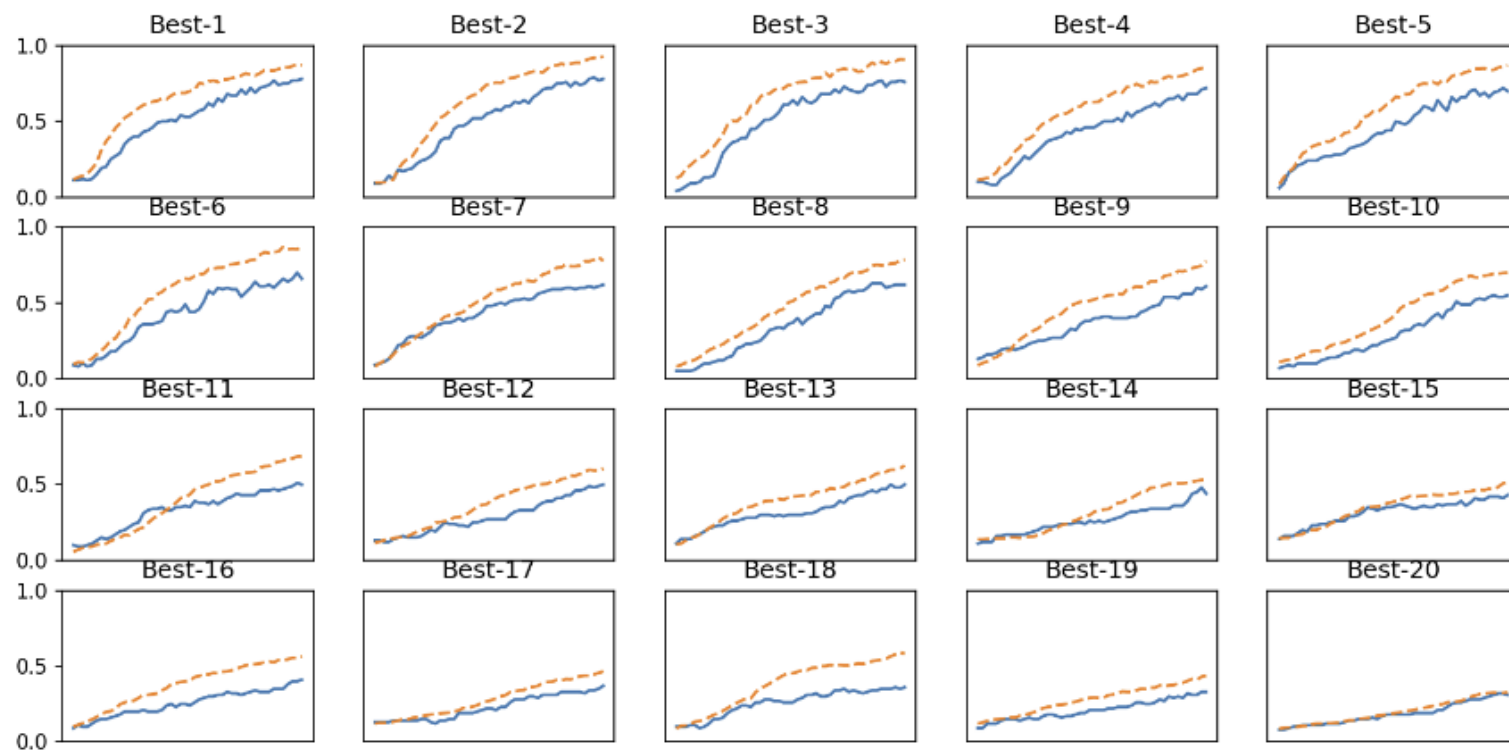
- **3단계**

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

5. 적절한 하이퍼파라미터 값 찾기

5.3 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. 결과값을 보면 다음과 같다.



가중치 감소 계수의 범위를 $10^{-8} \sim 10^{-4}$, 학습률의 범위를 $10^{-6} \sim 10^{-2}$ 로 설정

이제

이상입니다!