CHAPTER 6 학습 관련 기술들(3)

신경망(딥러닝) 학습의 효율과 정확도를 높이는 방법들

다른 매개변수 업그레이드 방법

매개변수 초기값 잘 주는 방법

배치 정규화

신경망 모델 오버피팅 방지하는 방법

좋은 하이퍼 파라미어 설정 방법

Contents

- CHAPTER 6 학습 관련 기술들
  - 6.1 매개변수 갱신
    - 6.1.1 모험가 이야기
    - 6.1.2 확률적 경사 하강법(SGD)
    - 6.1.3 SGD의 단점
    - 6.1.4 모멘텀
    - 6.1.5 AdaGrad
    - 6.1.6 Adam
    - 6.1.7 어느 갱신 방법을 이용할 것인가?
    - 6.1.8 MNIST 데이터셋으로 본 갱신 방법 비교
  - 6.2 가중치의 초깃값
    - 6.2.1 초깃값을 0으로 하면?
    - 6.2.2 은닉층의 활성화값 분포
    - 6.2.3 ReLU를 사용할 때의 가중치 초깃값
    - 6.2.4 MNIST 데이터셋으로 본 가중치 초깃값 비교
  - 6.3 배치 정규화
    - 6.3.1 배치 정규화 알고리즘
    - 6.3.2 배치 정규화의 효과
  - 6.4 바른 학습을 위해
    - 6.4.1 오버피팅
    - 6.4.2 가중치 감소
    - 6.4.3 드롭아웃
  - 6.5 적절한 하이퍼파라미터 값 찾기
    - 6.5.1 검증 데이터
    - 6.5.2 하이퍼파라미터 최적화
    - 6.5.3 하이퍼파라미터 최적화 구현하기
  - 6.6 정리

6.1.8 매개변수 갱신 방법 비교

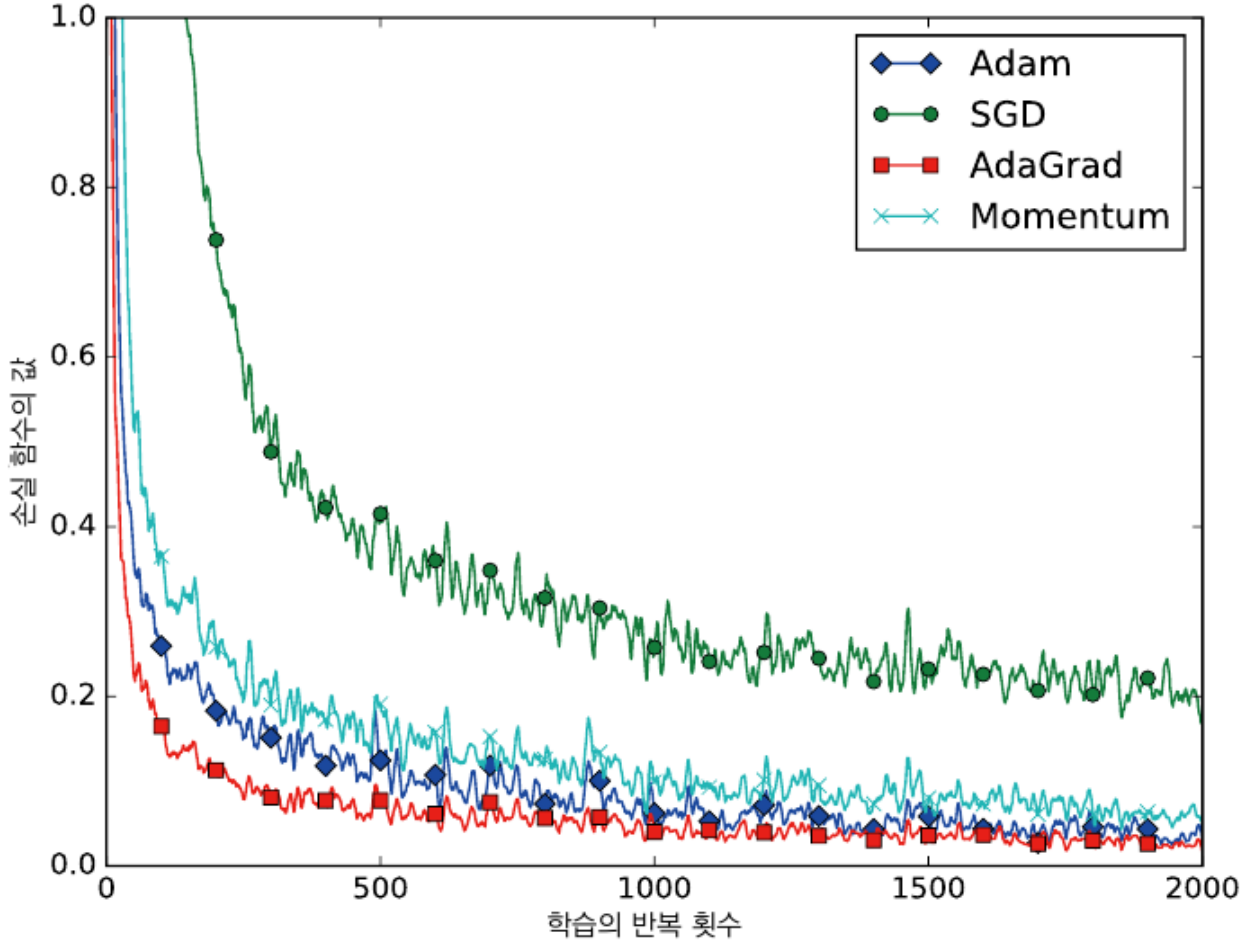
복습

MNIST 손글씨 숫자 인식을 대상으로 4 방법 탐색의 수렴 과정 비교

각 층이 100개 뉴런으로 구성된 5층 신경망에서 ReLU 활성화 함수 사용.
학습진도는 SGD가 가장 느리고 AdaGrad가 가장 빠름. 보통, SGD보다 다른 세 기법이 빠르게 학습하고, 정확도도 살짝 높다.

ch06/optimizer_compare_mnist.py

그림 6-9 MNIST 데이터셋에 대한 학습 진도 비교



6.2.4 가중치 초깃값 비교

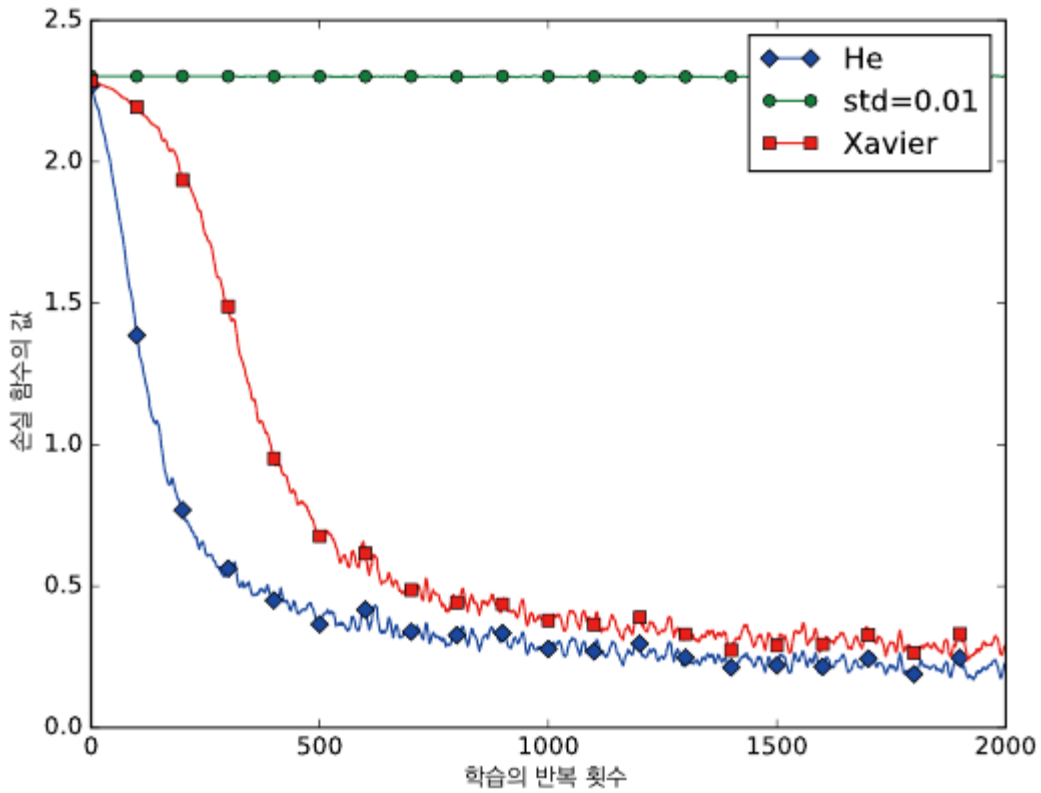
ch06/weight_init_compare.py

가중치 초깃값은 신경망 학습에 매우 중요하다.

가중치 초깃값에 따라 신경망 학습의 성패가

갈리는 경우가 많다.

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



이상의 실험 결과들을 바탕으로,

활성화 함수로 ReLU를 사용할 때는 He 초깃값을,

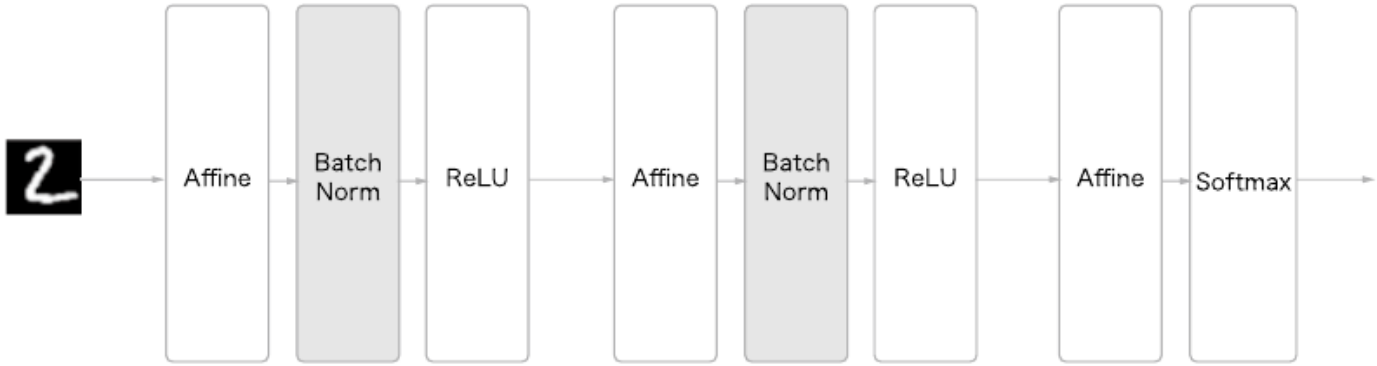
Sigmoid나 tanh를 사용할 때는 Xavier 초깃값을 사용을 권장.

6.3.1 배치 정규화 알고리즘

각 층이 활성화를 적당히 퍼뜨리도록 강제하는 방법

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

그림 6-16 배치 정규화를 사용한 신경망의 예



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

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

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

[식 6.7]

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

[식 6.8]

6.4 바른 학습을 위해

기계학습에서 오버피팅(Overfitting)이 문제가 되는 경우가 많다.

오버피팅이란, 신경망이 훈련 데이터에만 지나치게 적응되어 그 외의 데이터에는 제대로 대응하지 못하는 상태를 말한다. 훈련 데이터에는 포함되지 않은, 아직 보지 못한 데이터도 바르게 식별해 내는 모델이 바람직하다. *이러기 위해서는 학습데이터에 너무(over) 적합(fit)시키면 안 된다. 적당히 느슨하게 적합시켜야 한다.*

이 절에서는 오버피팅을 억제하는 기술인 '가중치 감소'와 '드롭아웃'에 대해 배운다.

일반화 성능(Generalization performance)라는 말을 자주 사용하는데,
학습데이터 외에, 새로운 데이터에 대해서도 잘 예측(분류)하는 모델을 일반화 성능이 좋은 모델이라고 한다. 이는 오버피팅이 안 된 모델이라 할 수 있다.
즉, "일반화 성능이 높다 = 오버피팅이 안 됐다."고 말할 수 있다.

6.4.1 오버피팅을 시켜보자 ^^

오버피팅은 다음 두 경우에 주로 발생한다.

- 매개변수가 많고 표현력이 높은(복잡한) 모델을 사용할 때
- 훈련데이터가 적은 경우

그러므로, 이 두가지 경우를 피할 수 있도록 노력해야 한다.

이번 절에서는, 일부러 이 두 요건을 충족하게 하게 학습하여, 정말 오버피팅이 발생하는지 확인해 보겠습니다.

6.4.1 오버피팅

먼저, 훈련 데이터를 조금만 사용하겠습니다. (전체 코드: ch06/overfit_weight_decay.py)

```
(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]
```

그리고, 매개변수가 많고 표현력이 높은(복잡한) 모델을 사용하겠습니다.

7층 신경망, 은닉층 뉴런 수 100개씩 (활성화함수 ReLU)

```
network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100,
100, 100], output_size=10)
```

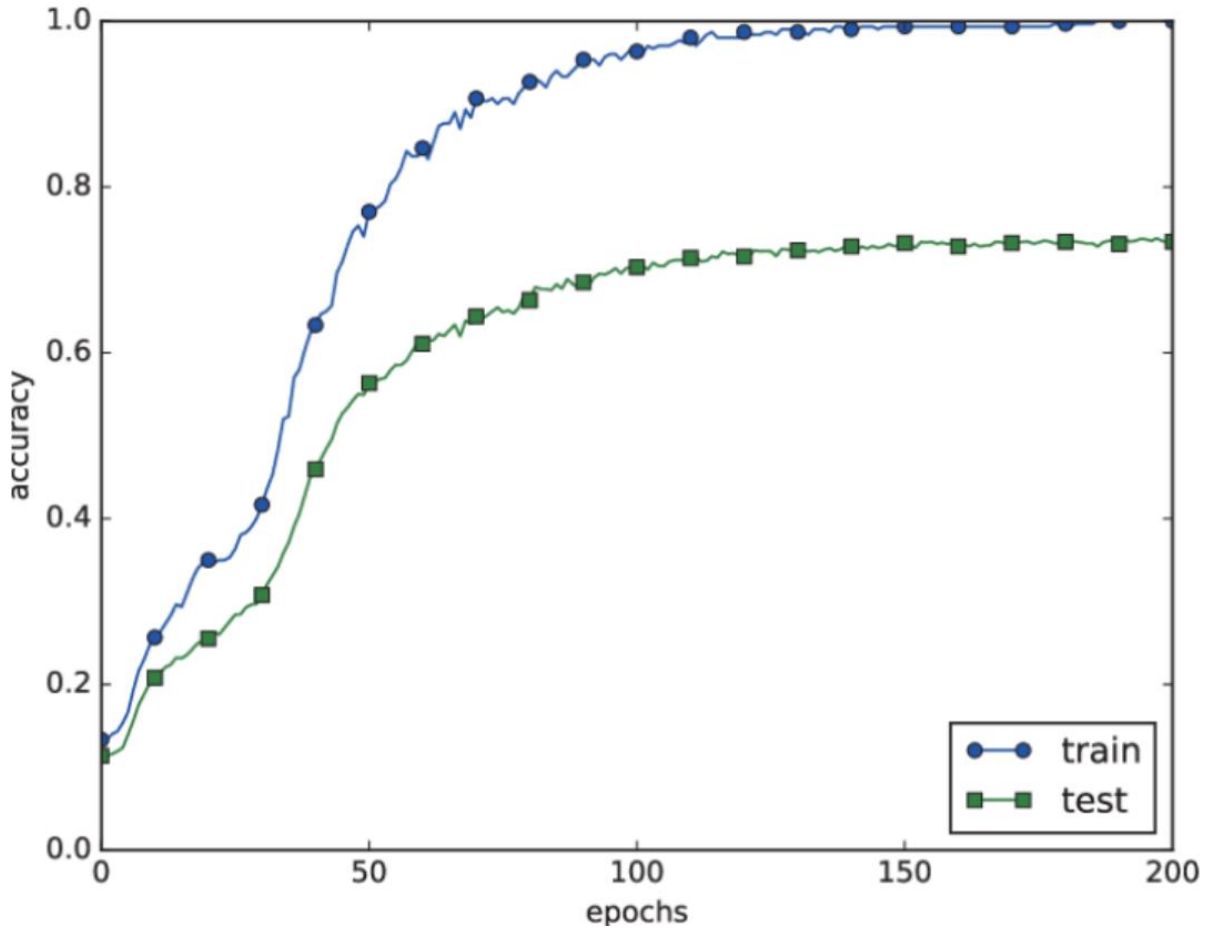
6.4.1 오버피팅

```
optimizer = SGD(lr=0.01) # 학습률이 0.01인 SGD로 매개변수 갱신
max_epochs = 201
train_size = x_train.shape[0]
batch_size = 100

train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = max(train_size / batch_size, 1)
epoch_cnt = 0
```

```
for i in range(1000000000):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

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

    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)

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

6.4.1 오버피팅

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



100 epoch을 지나서 부터, 훈련데이터는 거의 100% 정확하게 분류하지만, 시험 데이터는 그 정도 분류 못 함. 성능차이가 큼.

훈련데이터에만 적응해 버렸기 때문. 훈련 때 사용하지 않은 범용 데이터(시험 데이터)에 대해 제대로 분류(예측) 못하고 있는 것을 보여 줌.

6.4.2 가중치 감소

코드: common/multi_layer_net.py

가중치 감소weight decay : 오버피팅 억제를 위해 오래 전부터 많이 이용해온 방법 중 하나로, 학습 과정에서 크기가 큰 가중치에 대해서는 페널티를 부과하여 오버피팅을 억제하는 방법.

오버피팅은 가중치 매개변수의 값이 커서 발생하는 경우가 많다.

How? 가중치를 감소시키는 방법

: 신경망 학습의 목적은 손실함수 값을 줄이는 것이다. 가중치의 크기를 줄이기 위해 기존 손실함수에 가중치의 L2 노름(제곱 노름)을 더하여 새롭게 손실함수를 정의한다.

* 가중치의 L_2-norm인 W^2은 가중치의 크기이다.
* 편향 b의 L_2-norm은 손실함수에 더해 주지 않는다.

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

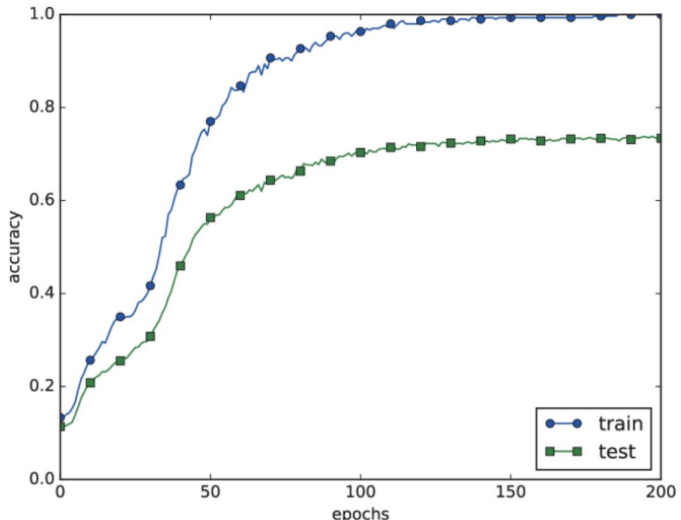
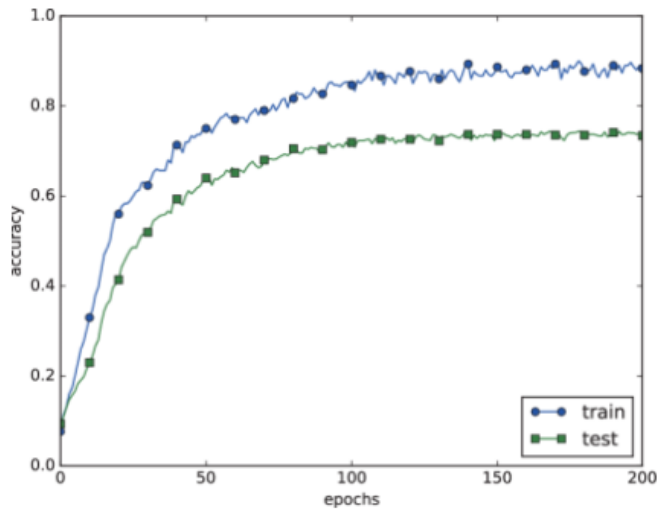


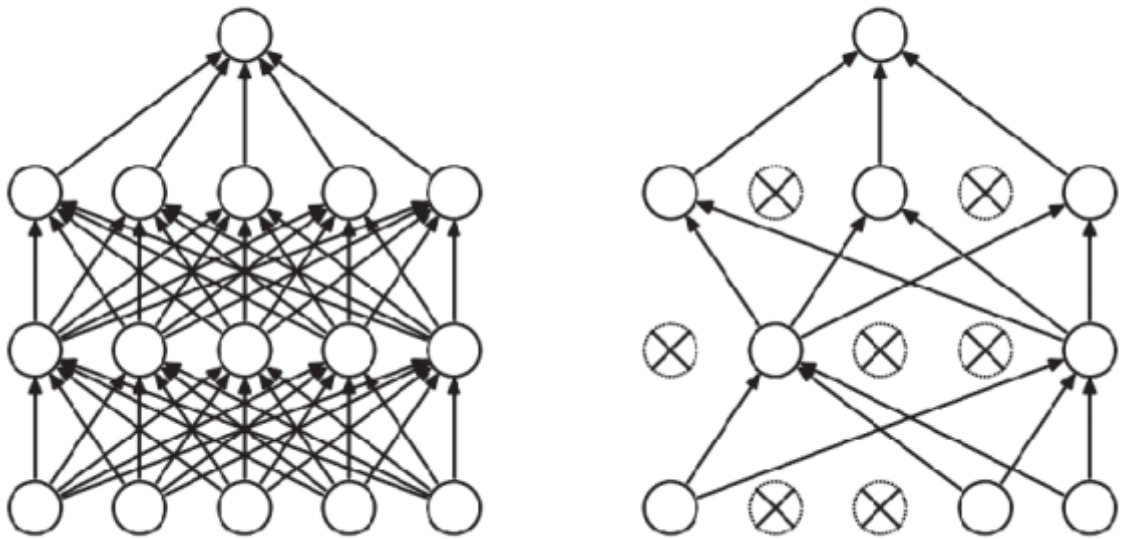
그림 6-21 가중치 감소를 이용한 훈련 데이터(train)와 시험 데이터(test)에 대한 정확도 추이



6.4.3 드롭아웃

드롭아웃 Dropout : 뉴런을 임의로 삭제하면서 학습하는 방법. 훈련(학습) 때 은닉층의 뉴런을 무직위로 골라 삭제하여 훈련함.
삭제된 뉴런은 [그림 6-22]와 같이 신호를 전달하지 않음. 큰 신경망에서는 가중치 감소보다 드롭아웃이 오버피팅 억제에 더 효과적이라 알려져 있음.
단. 시험(테스트) 때는 모든 뉴런을 사용함. 시험 때 각 뉴런의 출력 (forward) 에 훈련 때 삭제 안 한 비율을 곱해서 출력함.

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

6.4.3 드롭아웃

```
class Dropout:
    def __init__(self, dropout_ratio=0.5):
        self.dropout_ratio = dropout_ratio
        self.mask = None

    def forward(self, x, train_flg=True):
        if train_flg:
            self.mask = np.random.rand(*x.shape) > self.dropout_ratio
            return x * self.mask
        else:
            return x * (1.0 - self.dropout_ratio)

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

훈련시,
순전파 때 self.mask에 삭제한 뉴런을 False로 기록해 놓는다.
역전파 때 동작은 ReLU와 같다. 순전파 때 신호를 통과시킨 뉴런은 역전파 때도 신호를 그대로 통과시키고, 순전파 때 통과시키지 않은 뉴런은 역전파 때도 신호를 차단한다.

코드: common/trainer.py와 ch06/overfit_dropout.py

6.4.3 드롭아웃

코드: ch06/overfit_dropout.py

```
import os;  import 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)

# 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_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

# 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()
```

6.5. 적절한 하이퍼파라미터 값 찾기

신경망(Neural Network)에는 여러 하이퍼파라미터(Hyper-parameter)들이 등장한다.

각 층의 뉴런 수, 배치 크기, 매개 변수 학습률, 가중치 감소 계수 등이 있다.

이번 절에서는 하이퍼파라미터 값을 효율적으로 탐색하는 방법을 공부한다.

검증 데이터는 하이퍼파라미터를 결정할 때 사용하는 데이터를 말한다.

그리고, 하이퍼파라미터를 결정하는 Randomized Grid Search 방법에 대해 설명하고, 구현하고, 결과를 살펴 본다.

6.5.1 검증 데이터

하이퍼파라미터를 조정(결정)할 때, 하이퍼파라미터 전용 확인 데이터가 필요하다.
하이퍼파라미터 조정용 데이터를 일반적으로 검증 데이터validation data라고 부른다. 검증 데이터는 보통 훈련데이터의 일부를 사용한다.

훈련 데이터: 매개변수 학습
검증 데이터: 하아퍼파라미터 성능평가 > 최선의 하이퍼파라미터 결정
시험 데이터: 신경망의 범용 성능 평가

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

# 훈련 데이터를 뒤섞는다.
x_train, t_train = shuffle_dataset(x_train, t_train)

# 20%를 검증 데이터로 분할
validation_rate = 0.20
validation_num = int(x_train.shape[0] * validation_rate)

x_val = x_train[:validation_num]
t_val = t_train[:validation_num]
x_train = x_train[validation_num:]
t_train = t_train[validation_num:]
```

shuffle_dataset 함수는 common/util.py 파일에 구현되어 있음.

6.5.2 하이퍼파라미터 최적화

최선의 하이퍼파라미터를 탐색하는(찾는) 가장 좋은 방법은, 최선(최적)의 하이퍼파라미터 값이 존재하는 범위를 조금씩 좁혀가는 것이다.

범위를 조금씩 좁히려면, 우선 대략적인 범위를 설정하고 그 범위 안에서 무작위로 하이퍼파라미터를 선택(샘플링)한 후, 그 때의 신경망의 정확도를 평가한다. 이를 여러 번 반복하여, 그 중 가장 좋은 하이퍼파라미터를 선정한다.

그 다음, 그 하이퍼파라미터 주변으로 다시 범위를 설정하고 샘플링과 평가를 몇 번 더 반복한 후 결정한다.

0단계 : 하이퍼파라미터 값의 범위를 설정.

1단계 : 설정된 범위에서 하이퍼파라미터의 값을 무작위로 추출.

2단계 : 1단계에서 샘플링한 하이퍼파라미터 값을 사용하여 학습하고, 검증 데이터로 정확도를 평가 (단, 효율적인 계산을 위해 에폭은 작게 설정해도 괜찮다).

3단계 : 1단계와 2단계를 특정 횟수(100회 등) 반복하며, 그 정확도의 결과를 보고 하이퍼파라미터의 범위를 좁힌다.

6.5.3 하이퍼파라미터 최적화 구현하기

MNIST 데이터셋을 사용하여 하이퍼파라미터를 최적화해 보자.
하이퍼파라미터 중 학습률과 가중치 감소 계수만 결정해 보겠습니다. (다른 하이퍼 파라미터는 defaul로 사용)
실험에서는, 이 데이터셋에 대한 최선의 학습률(learning rate)과 가중치 감소 계수(손실함수에 더해진 가중치 제곱의 합에 곱해져, 가중치의 감소의 세기를 조절하는 계수)를 탐색(결정)해 보겠습니다.

가중치 감소 계수를 10^{-8} ~ 10^{-4}, 학습률은 10^{-6}~10^{-2} 범위에서 시작하겠습니다. (이 범위는 감으로 찍은 것임)
이는 다음과 같이 구현할 수 있습니다.

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

6.5.3 하이퍼파라미터 최적화 구현하기

코드: ch06/hyperparameter_optimization.py 파일 참조

```
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
```

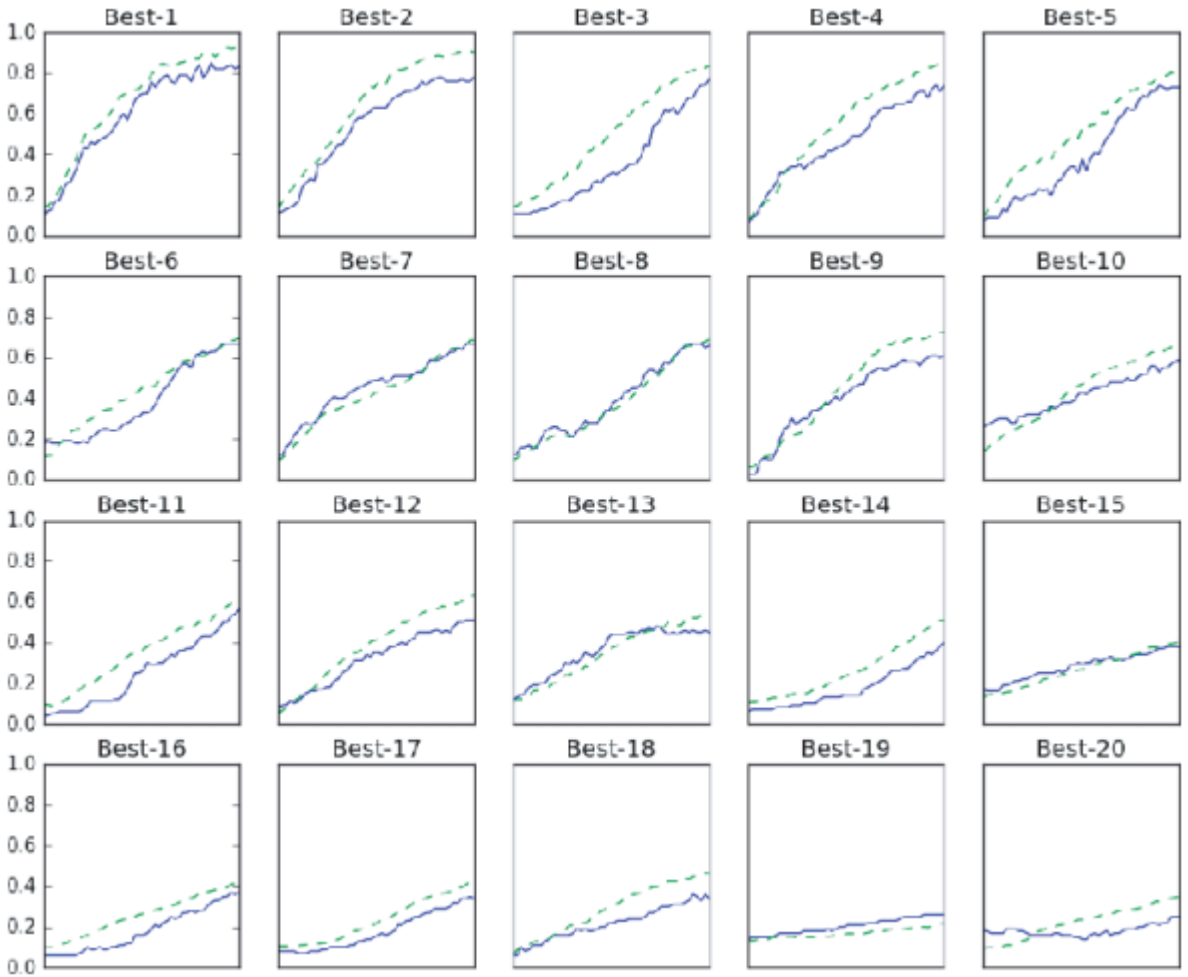
6.5.3 하이퍼파라미터 최적화 구현하기

코드: ch06/hyperparameter_optimization.py 파일 참조

```
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
```

6.5.3 하이퍼파라미터 최적화 구현하기

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




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

학습률은 0.001~0.01, 가중치 감소 계수는 10-8~10-6 정도에서 학습이 잘 되는 것으로 보고, 이 (좁혀진) 범위에서 한 번 더 샘플링하여 평가하는 작업을 수행할 것을 권장함.

이렇게 하이퍼파라미터 값의 범위를 몇번 더 좁혀가다가, 어느 시점에서 그만하고, 최선이다 싶은 하이퍼파라미터를 결정합니다.

6.6 정리

이번 장에서 배운 내용

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