6장 학습 관련 기술들

이번 장에서는 신경망 학습의 핵심 개념들을 만나봅니다. 이번 장에서 다룰 주제는 가중치 매개변수의 최적값을 탐색하는 최적화 방법, 가중치 매개변수 초깃값, 하이퍼파라미터 설정 방법 등, 모두가 신경망 학습에서 중요한 주제입니다. 오버피팅의 대응책인 가중치 감소와 드롭아웃 등의 정규화 방법도 간략히 설명하고 구현해봅니다. 마지막으로 최근 많은 연구에서 사용하는 배치 정규화도 짧게 알아봅니다. 이번 장에서 설명하는 기법을 이용하면 신경망(딥러닝) 학습의 효율과 정확도를 높일 수 있습니다.

Customized by Gil-Jin Jang, April 8, 2021

파일 설명

파일명	파일 용도	관련 절	페이지
batch_norm_gradient_check.py	배치 정규화를 구현한 신경망의 오차역전파법 방식의 기울기 계산이 정확한지 확인합니다(기울기 확인).		
batch_norm_test.py	MNIST 데이터셋 학습에 배치 정규화를 적용해봅니다.	6.3.2 배치 정규화의 효과	212
hyperparameter_optimization.py	무작위로 추출한 값부터 시작하여 두 하이퍼파라미터(가중치 감소 계수, 학습률)를 최적화해봅니다.	6.5.3 하이퍼파라미터 최적화 구현하기	224
optimizer_compare_mnist.py	SGD, 모멘텀, AdaGrad, Adam의 학습 속도를 비교합니다.	6.1.8 MNIST 데이터셋으로 본 갱신 방법 비교	201
optimizer_compare_naive.py	SGD, 모멘텀, AdaGrad, Adam의 학습 패턴을 비교합니다.	6.1.7 어느 갱신 방법을 이용할 것인가?	200
overfit_dropout.py	일부러 오버피팅을 일으킨 후 드롭아웃(dropout)의 효과를 관찰합니다.	6.4.3 드롭아웃	219
overfit_weight_decay.py	일부러 오버피팅을 일으킨 후 가중치 감소(weight_decay)의 효과를 관찰합니다.	6.4.1 오버피팅	215
weight_init_activation_histogram.py	활성화 함수로 시그모이드 함수를 사용하는 5층 신경망에 무작위로 생성한 입력 데이터를 흘리며 각 층의 활성화값 분포를 히스토그램으로 그려봅니다.	6.2.2 은닉층의 활성화값 분포	203
weight_init_compare.py	가중치 초깃값(std=0.01, He, Xavier)에 따른 학습 속도를 비교합니다.	6.2.4 MNIST 데이터셋으로 본 가중치 초깃값 비교	209

목차

6.1 매개변수 갱신

최적화 : 손실 함수의 값을 가능한 낮추는 매개변수를 찾음. 최적 매개변수를 찾는 문제를 푸는 것.

확률적 경사 하강법(SGD) : 매개변수 기울기를 구해 기울어진 방향으로 매개변수 값을 갱신하여 점점 최적의 매개변수로 다가가는 방법

6.1.1 모험가 이야기

SGD의 전략: 지금 서있는 장소에서 가장 크게 기울어진 방향으로 가는 것.

6.1.2 확률적 경사 하강법(SGD)

식 6.1 SGD

$$W := W - \eta \frac{\partial L}{\partial W}$$

W : 갱신할매개변수
$\frac{\partial L}{\partial W}$: 손실함수의기울기
η : 학습률, 미리정해서사용

:= 우변의 값으로 좌변의 값을 갱신

SGD 파이썬 클래스 구현

In [1]:
```
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]
```

lr : 학습률, learning rate. 인스턴스 변수로 유지

update(params, grads) : SGD 과정에서 반복해서 호출됨

params : 딕셔너리 변수. 가중치 매개변수 저장됨. 예시 params['W1']

grads : 딕셔너리 변수. 기울기가 저장됨. 예시 grads['W1']

신경망 매개변수 진행 의사코드

```
network = TwoLayerNet(...)
optimizer = SGD() ###
for i in range(10000):
    ...
    x_batch, t_batch = get_mini_batch(...) # 미니배치
    grads = network.gradient(x_batch, t_batch)
    params = network.params
    optimizer.update(params, grads) ###
    ...
```

optimizer : 변수. 뜻은 최적화를 행하는 자

매개변수 갱신은 optimizer가 책임지고 수행. optimizer에 매개변수와 기울기 정보만 넘기면 됨

최적화를 담당하는 클래스를 분리 구현하면 기능을 모듈화하기 좋음

모멘텀 최적화 기법 역시 update(params, grads)라는 공통의 메서드를 갖도록 구현

optimizer = SGD() 문장을 optimizer = Momentum()으로만 변경하면 됨

Lasagne 딥러닝 프레임워크는 최적화 기법을 다음 함수들로 정리함

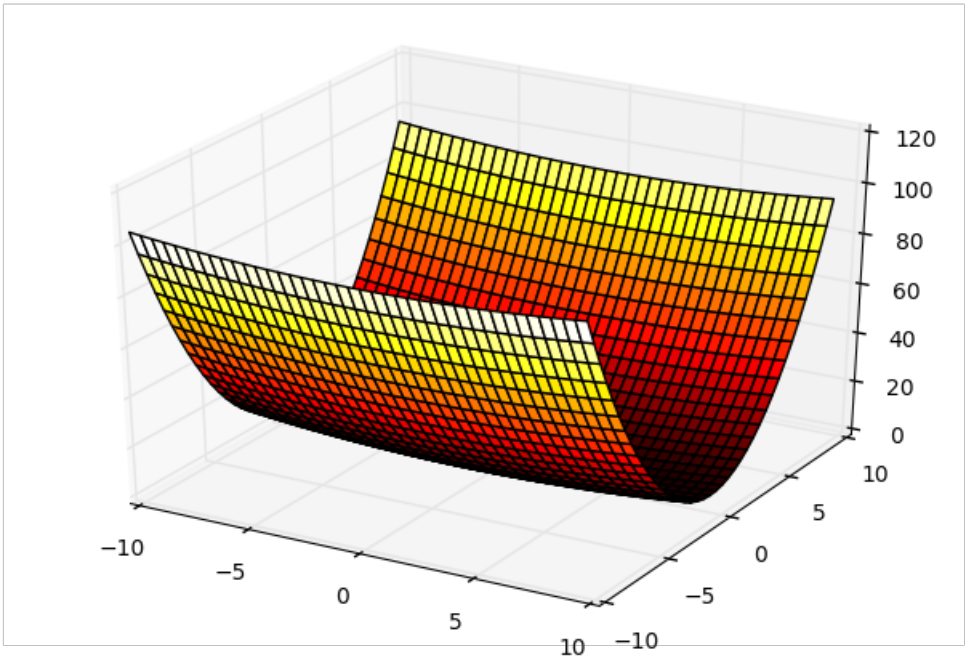
https://github.com/Lasagne/Lasagne/blob/master/lasagne/updates.py (https://github.com/Lasagne/Lasagne/blob/master/lasagne/updates.py)

6.1.3 SGD의 단점

식 6.2 SGD

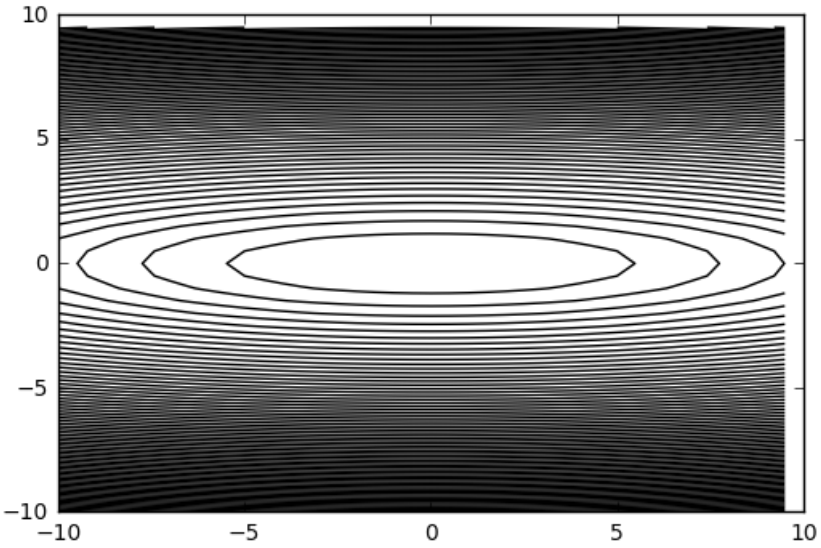
$$f(x, y) = \frac{1}{20}x^2 + y^2$$

In [2]:
```
# 그림 6-1 f(x, y) = (1/20) * x**2 + y**2 그래프
# 3차원 참고주소: https://www.datascienceschool.net/view-notebook/6e71dbff254542d9b0a054a7c98b34ec/
%matplotlib inline
import numpy as np
import matplotlib.pylab as plt
from mpl_toolkits.mplot3d import Axes3D
X = np.arange(-10, 10, 0.5)
Y = np.arange(-10, 10, 0.5)
XX, YY = np.meshgrid(X, Y)
ZZ = (1 / 20) * XX**2 + YY**2

fig = plt.figure()
ax = Axes3D(fig)
ax.plot_surface(XX, YY, ZZ, rstride=1, cstride=1, cmap='hot');
```



In [3]:
```
# 그림 6-1 f(x, y) = (1/20) * x**2 + y**2 등고선
plt.contour(XX, YY, ZZ, 100, colors='k')
plt.ylim(-10, 10)
plt.xlim(-10, 10)
```

Out[3]: (-10, 10)



식 6.2 기울기 특징

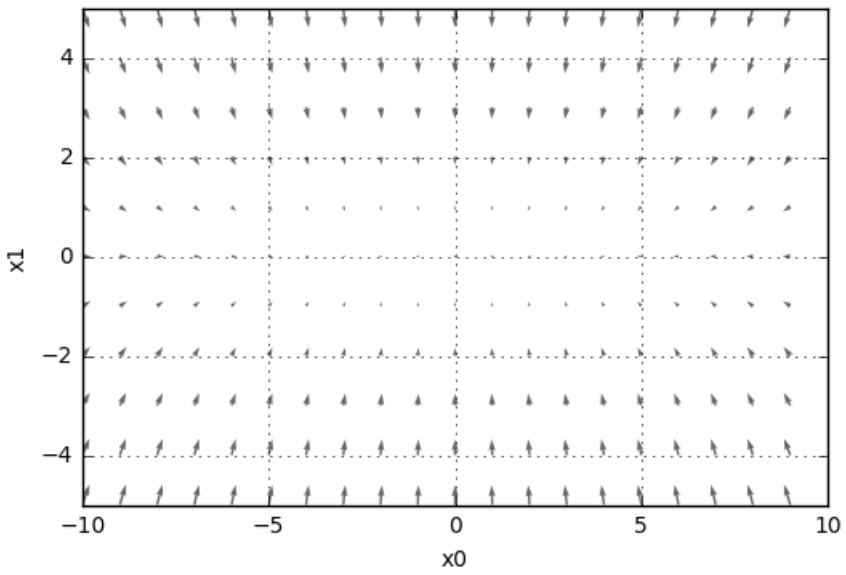
y축 방향은 가파른데 x 축 방향은 완만함.

기울기의 대부분은 (0,0) 방향을 가리키지 않음.

In [4]:
```
def _numerical_gradient_no_batch(f, x):
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x) # x와 형상이 같은 배열을 생성

    for idx in range(x.size):
        tmp_val = x[idx]

        # f(x+h) 계산
        x[idx] = float(tmp_val) + h
        fxh1 = f(x)

        # f(x-h) 계산
        x[idx] = tmp_val - h
        fxh2 = f(x)

        grad[idx] = (fxh1 - fxh2) / (2*h)
        x[idx] = tmp_val # 값 복원

    return grad
```

In [5]:
```
# 그림 6-2 f(x, y) = (1/20) * x**2 + y**2 의 기울기
# https://github.com/WegraLee/deep-learning-from-scratch/blob/master/ch04/gradient_2d.py 소스 참고
from mpl_toolkits.mplot3d import Axes3D

def numerical_gradient(f, X):
    if X.ndim == 1:
        return _numerical_gradient_no_batch(f, X)
    else:
        grad = np.zeros_like(X)

        for idx, x in enumerate(X):
            grad[idx] = _numerical_gradient_no_batch(f, x)

        return grad

def function_2(x):
    if x.ndim == 1:
        return np.sum(x**2)
    else:
        return np.sum(x**2, axis=1)

x0 = np.arange(-10, 10, 1)
x1 = np.arange(-10, 10, 1)
X, Y = np.meshgrid(x0, x1)

X = X.flatten()
Y = Y.flatten()

grad = numerical_gradient(function_2, np.array([(1/(20**0.5))*X, Y]) )

plt.figure()
plt.quiver(X, Y, -grad[0], -grad[1],  angles="xy",color="#666666")#,headwidth=10,scale=40,color="#444444")
plt.xlim([-10, 10])
plt.ylim([-5, 5])
plt.xlabel('x0')
plt.ylabel('x1')
plt.grid()
plt.legend()
plt.draw()
plt.show()
```

```
C:\Users\RyanShin\Anaconda3\lib\site-packages\matplotlib\axes\_axes.py:531: UserWarning: No labelled objects found. Use label='...' kwarg on individual plots.
  warnings.warn("No labelled objects found. "
```



SGD 단점 : 비등방성(anisotropy) 함수(방향에 따라 기울기가 달라지는 함수)에서 탐색경로가 비효율적.

6.1.4 모멘텀

모멘텀: Momentum, 운동량

식 6.3

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

식 6.4

$$W := W + v$$

W : 갱신할매개변수
$\frac{\partial L}{\partial W}$: 손실함수의기울기
η : 학습률, 미리정해서사용

v : 물리에서 말하는 속도(velocity)

식 6.3 : 기울기 방향으로 힘을 받아 물체가 가속된다는 물리 법칙을 나타냄

식 6.4 : 모멘텀에 따라 공이 그릇의 바닥을 구르는 듯한 움직임을 보임

모멘텀의 구현

In [6]:
```
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]
```

v: 물체의 속도

v는 초기화 때는 아무것도 담지 않고, update가 처음 호출될 때 같은 구조의 데이터를 딕셔너리 변수로 저장

모멘텀의 갱신경로는 공이 그릇 바닥을 구르듯 움직임

SGD와 비교하면 지그재그 정도가 덜함

6.1.5 AdaGrad

학습률 감소(learning rate decay) : 학습을 진행하면서 학습률을 점차 줄여가는 방법

개별 매개변수에 적응적으로(adative) 학습률을 조정하면서 학습을 진행

식 6.5

$$h := h + \frac{\partial L}{\partial W} \odot \frac{\partial L}{\partial W}$$

식 6.6

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

W : 갱신할매개변수
$\frac{\partial L}{\partial W}$: 손실함수의기울기
η : 학습률, 미리정해서사용

h : 기존 기울기 값을 제곱하여 계속 더해줌

매개변수를 갱신할 때 1/sqrt(h)을 곱해 학습률을 조정

AdaGrad는 학습을 진행할 수록 갱신 강도가 약해짐.

RMSProp

지수이동평균(Exponential Moving Average, EMA)를 이용하여 먼 과거의 기울기를 서서히 잊고 새로운 기울기 정보를 크게 반영.

AdaGrad의 구현

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

마지막 줄에서 1e-7이라는 작은 값을 더하는 부분이 0으로 나누는 사태를 막음

대부분의 딥러닝 프레임워크에서 이 값도 인수로 설정 가능

처음에는 크게 움직이지만 갱신 정도가 작아지도록 조정됨

6.1.6 Adam

모멘텀: 공이 그릇을 구르는 듯한 물리 법칙에 따르는 움직임

AdaGrad: 매개변수의 원소마다 적응적으로 갱신 정도를 조정

Adam: 모멘텀과 Adagrad 기법을 융합

매개변수 공간을 효율적으로 탐색. 하이터파라미터의 '편향 보정'이 진행됨

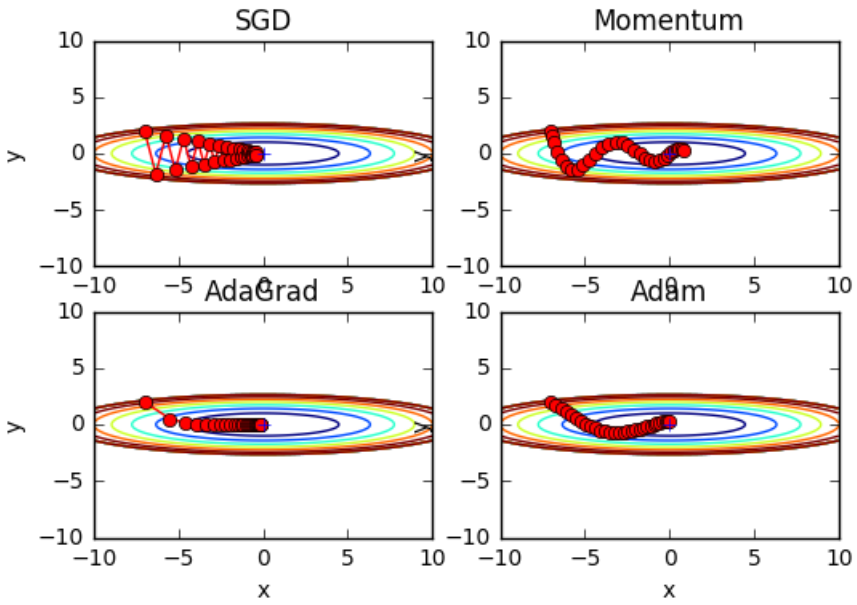
Adam의 구현

In [8]:
```
# https://github.com/WegraLee/deep-learning-from-scratch/blob/master/common/optimizer.py 참고
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.iter) / (1.0 - self.beta1**self.iter)


        for key in params.keys():
            #self.m[key] = self.beta1*self.m[key] + (1-self.beta1)*grads[key]
            #self.v[key] = self.beta2*self.v[key] + (1-self.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(self.v[key]) + 1e-7)

            #unbias_m += (1 - self.beta1) * (grads[key] - self.m[key]) # correct bias
            #unbisa_b += (1 - self.beta2) * (grads[key]*grads[key] - self.v[key]) # correct bi
as
            #params[key] += self.lr * unbias_m / (np.sqrt(unbisa_b) + 1e-7)
```

6.1.7 어느 갱신 방법을 이용할 것인가?

네 기법의 결과를 비교

In [9]:
```
# https://github.com/WegraLee/deep-learning-from-scratch/blob/master/ch06/optimizer_compare_naive.py 참고
# coding: utf-8
import numpy as np
import matplotlib.pyplot as plt
from collections import OrderedDict

def f(x, y):
    return x**2 / 20.0 + y**2

def df(x, y):
    return x / 10.0, 2.0*y

init_pos = (-7.0, 2.0)
params = {}
params['x'], params['y'] = init_pos[0], init_pos[1]
grads = {}
grads['x'], grads['y'] = 0, 0


optimizers = OrderedDict()
optimizers["SGD"] = SGD(lr=0.95)
optimizers["Momentum"] = Momentum(lr=0.1)
optimizers["AdaGrad"] = AdaGrad(lr=1.5)
optimizers["Adam"] = Adam(lr=0.3)

idx = 1

for key in optimizers:
    optimizer = optimizers[key]
    x_history = []
    y_history = []
    params['x'], params['y'] = init_pos[0], init_pos[1]

    for i in range(30):
        x_history.append(params['x'])
        y_history.append(params['y'])

        grads['x'], grads['y'] = df(params['x'], params['y'])
        optimizer.update(params, grads)


    x = np.arange(-10, 10, 0.01)
    y = np.arange(-5, 5, 0.01)

    X, Y = np.meshgrid(x, y)
    Z = f(X, Y)

    # 외곽선 단순화
    mask = Z > 7
    Z[mask] = 0

    # 그래프 그리기
    plt.subplot(2, 2, idx)
    idx += 1
    plt.plot(x_history, y_history, 'o-', color="red")
    plt.contour(X, Y, Z)
    plt.ylim(-10, 10)
    plt.xlim(-10, 10)
    plt.plot(0, 0, '+')
    #colorbar()
    #spring()
    plt.title(key)
    plt.xlabel("x")
    plt.ylabel("y")

plt.show()
```



문제가 무엇이냐에 따라 사용할 기법이 달라짐

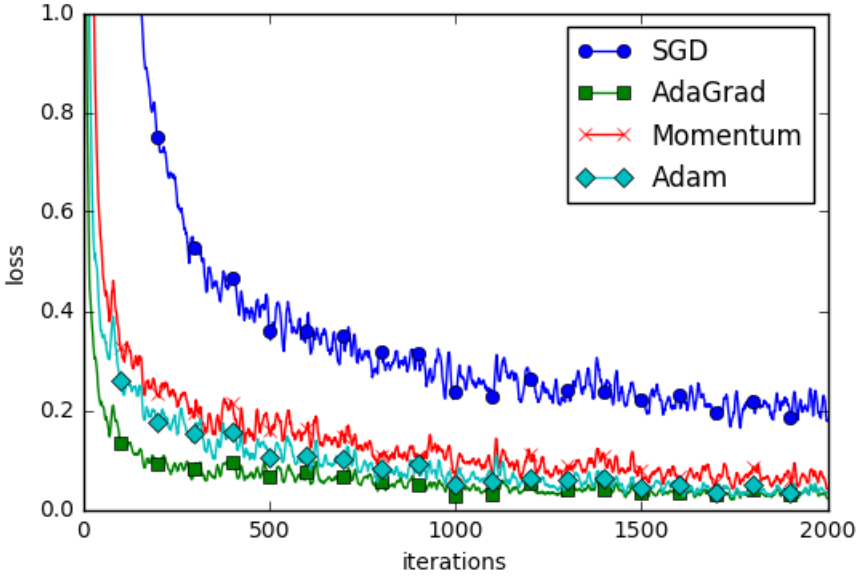
하이퍼 파라미터를 어떻게 설정하느냐에 따라서 결과도 바뀜

모든 문제에서 항상 뛰어난 기법은 아직 없음

이 책에서는 SGD, Adam을 많이 사용.

In [10]:
```
# https://github.com/WegraLee/deep-learning-from-scratch/blob/master/ch06/optimizer_compare_mnist.py 참고
# 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()
plt.show()
```

```
==========iteration:0==========
SGD:2.34085772575
AdaGrad:2.25657183172
Momentum:2.34769007692
Adam:2.22245332649
==========iteration:100==========
SGD:1.49464070056
AdaGrad:0.126855431365
Momentum:0.289574206257
Adam:0.240915687079
==========iteration:200==========
SGD:0.787281428925
AdaGrad:0.0856404792796
Momentum:0.243910289024
Adam:0.164698586307
==========iteration:300==========
SGD:0.569256864908
AdaGrad:0.116224924379
Momentum:0.220996306945
Adam:0.204483505084
==========iteration:400==========
SGD:0.392445324199
AdaGrad:0.068491966373
Momentum:0.156081337038
Adam:0.153620743223
==========iteration:500==========
SGD:0.372587776549
AdaGrad:0.0458588856307
Momentum:0.106137811039
Adam:0.0634062457435
==========iteration:600==========
SGD:0.385125600532
AdaGrad:0.110250011397
Momentum:0.241495830798
Adam:0.175897435735
==========iteration:700==========
SGD:0.337778819012
AdaGrad:0.111162440781
Momentum:0.205493288828
Adam:0.191697396219
==========iteration:800==========
SGD:0.387110218454
AdaGrad:0.0843285723211
Momentum:0.100320167247
Adam:0.0870536704711
==========iteration:900==========
SGD:0.276739171722
AdaGrad:0.0488620001706
Momentum:0.0904530038876
Adam:0.0973251839323
==========iteration:1000==========
SGD:0.234908385797
AdaGrad:0.0191196100601
Momentum:0.0682643784957
Adam:0.0540995924508
==========iteration:1100==========
SGD:0.251028047392
AdaGrad:0.0584517138455
Momentum:0.0646557945034
Adam:0.0881911335252
==========iteration:1200==========
SGD:0.29385994219
AdaGrad:0.0334386973774
Momentum:0.136223982857
Adam:0.0246669623397
==========iteration:1300==========
SGD:0.244678946951
AdaGrad:0.0578839181775
Momentum:0.0749945296491
```



각 층이 100개의 뉴런으로 구성된 5층 신경망에서 ReLU를 활성화 함수로 사용해 측정

하이퍼파라미터인 학습률과 신경망의 구조(층 깊이 등)에 따라 결과가 달라짐

일반적으로 SGD보다 다른 세 기법이 빠르게 학습하고, 때로는 최종 정확도도 높음

6.2 가중치의 초기값

가중치의 초기값을 무엇으로 설정하느냐가 신경망 학습의 성패를 가름

6.2.1 초기값을 0으로 하면?

가중치 감소(weight decay)

- 가중치 매개변수 값이 작아지도록 학습하는 방법.
- 가중치 값을 작게 하여 오버피팅이 일어나지 않음.

초기값을 작게 하기 위해 0.01 * np.random.randn(10,100) 처럼 정규분포로 생성된 값에 0.01을 곱함

초기값을 모두 0으로 해서는 안되는 이유?

오차역전법에서 모든 가중치의 값이 똑같이 갱신되기 때문

- 순전파: 입력파의 가중치가 0이기 때문에 두 번째 층의 뉴런에 모두 같은 값이 전달
- 역전파: 가중치가 모두 똑같이 갱신

6.2.2 은닉층의 활성화값 분포

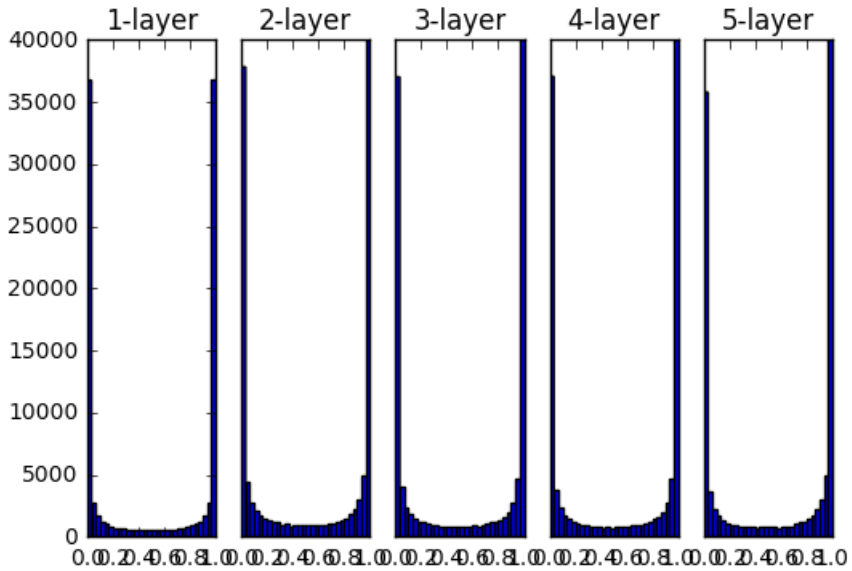
가중치의 초기값에 따라 은닉층 활성화 값들이 어떻게 변화는지 실험

각 층의 활성화값 분포를 히스토그램으로 그림

In [11]:
```
# https://github.com/WegraLee/deep-learning-from-scratch/blob/master/ch06/weight_init_activation_histogram.py 참고
# 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

def get_activation(hidden_layer_size, x, w, a_func=sigmoid):
    for i in range(hidden_layer_size):
        if i != 0:
            x = activations[i-1]

        a = np.dot(x, w)

        # 활성화 함수도 바꿔가며 실험해보자 !
        z = a_func(a)
        # z = ReLU(a)
        # z = tanh(a)

        activations[i] = z
    return activations

# 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)

z = sigmoid
# z = ReLU
# z = tanh

activations = get_activation(hidden_layer_size, x, w, z)
```

표준편차가 1인 정규분포의 활성화값들의 분포

In [12]:
```
# 히스토그램 그리기
def get_histogram(activations):
    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()

get_histogram(activations)
```

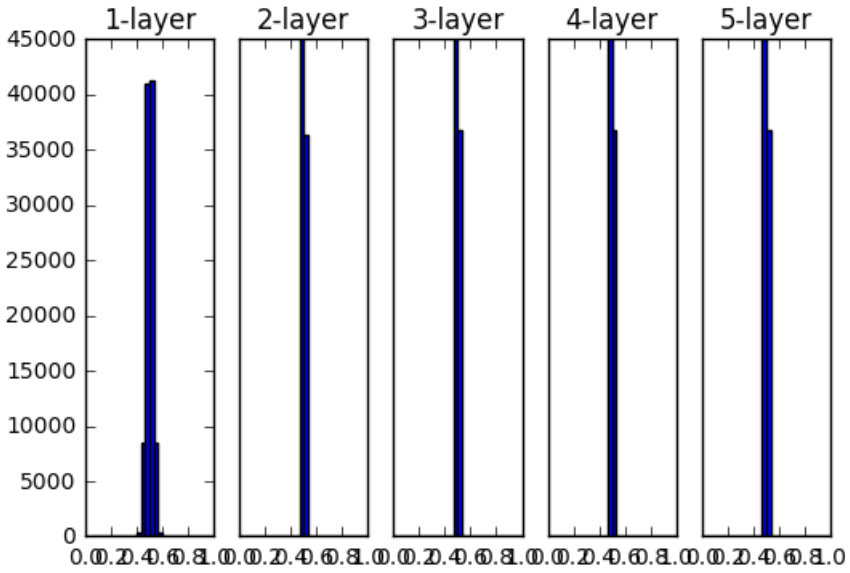


기울기 소실(gradient vanishing)

데이터가 0과 1에 치우쳐 분포하게 되면 역전파 기울기 값이 점점 작아지다 사라짐

가중치의 표준편차를 0.01로 변경

In [13]:
```
w = np.random.randn(node_num, node_num) * 0.01
activations = get_activation(hidden_layer_size, x, w, z)
get_histogram(activations)
```



0.5 부근에 집중. 활성화값들이 치우쳤다는 것은 표현력 관점에서 큰 문제

표현력을 제한: 예를 들어 뉴런 100개가 거의 같은 값을 출력한다면 뉴런 1개짜리와 별반 다를게 없음

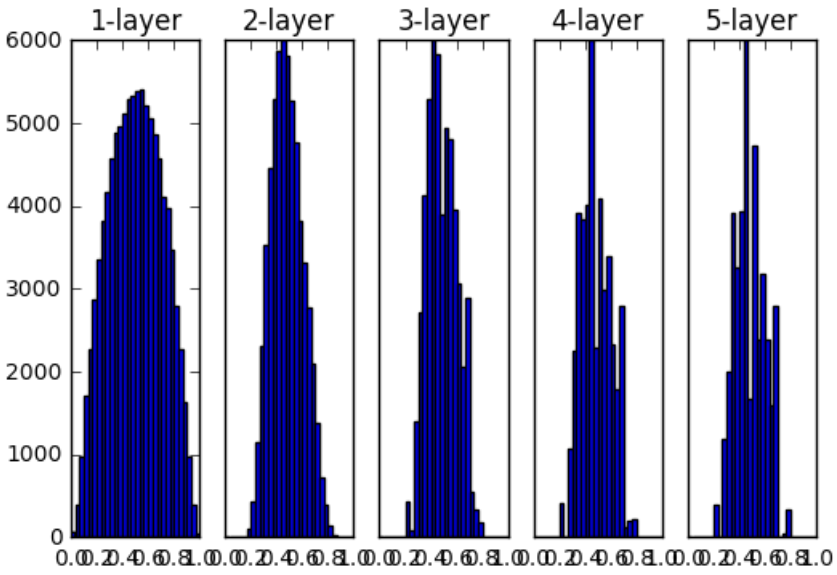
Xavier 초기값

앞 계층의 노드가 n개라면 표준편차가 1 / np.sqrt(n)인 정규분포를 사용

사비에르 논문은 앞 층의 노드 수 외에 다음 출력 노드 수도 고려한 설정 값 제안.

카페 등의 프레임워크는 앞층의 입력 노드만으로 계산하도록 단순화.

In [14]:
```
# Xavier 초기값
w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)
activations = get_activation(hidden_layer_size, x, w, z)
get_histogram(activations)
```



층이 깊어지면서 형태가 다소 일그러지지만, 넓게 분포됨

시그모이드 함수의 표현력도 제한받지 않고 학습이 효율적으로 이뤄질 것

6.2.3 ReLU를 사용할 때의 가중치 초기값

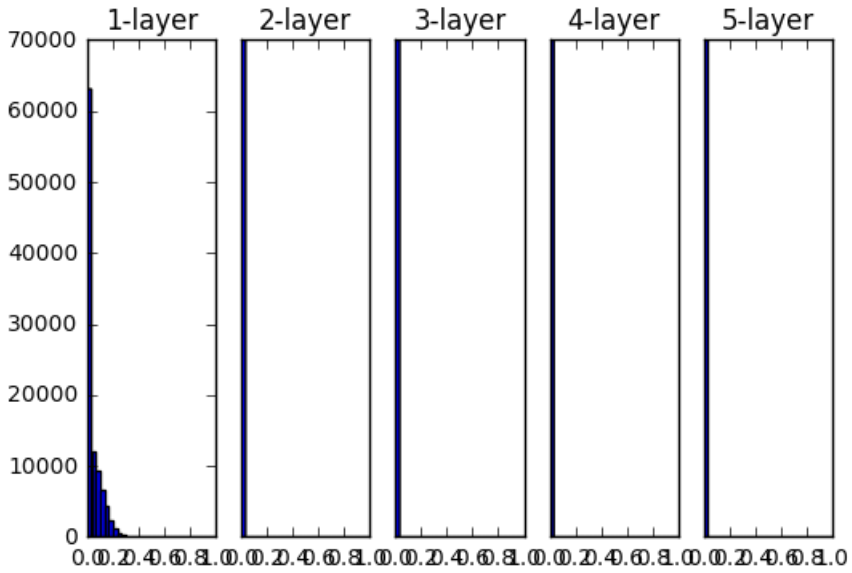
sigmoid, tanh는 좌우 대칭이라 Xavier 초기값이 적당

He 초기값

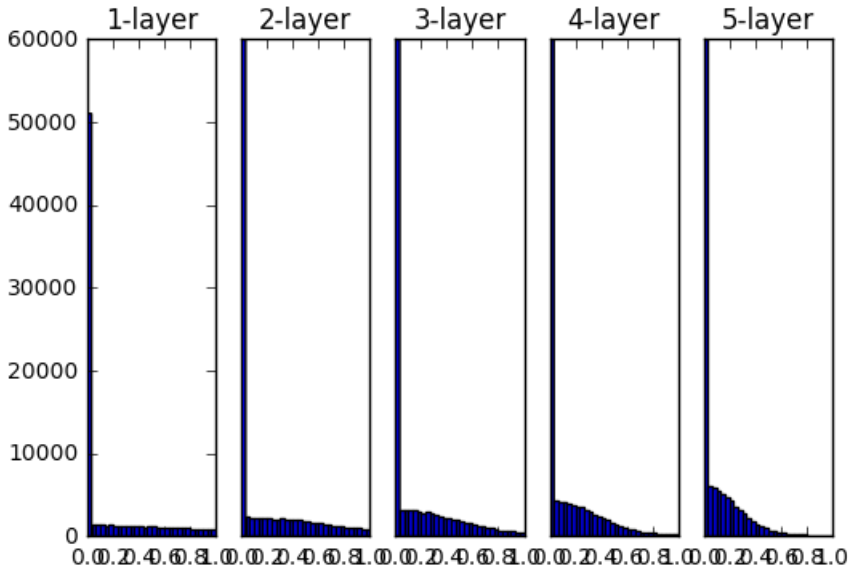
ReLu에 특화된 초기값.

앞 계층의 노드가 n일 때 표준편차가 2 / np.sqrt(n)인 정규분포를 사용

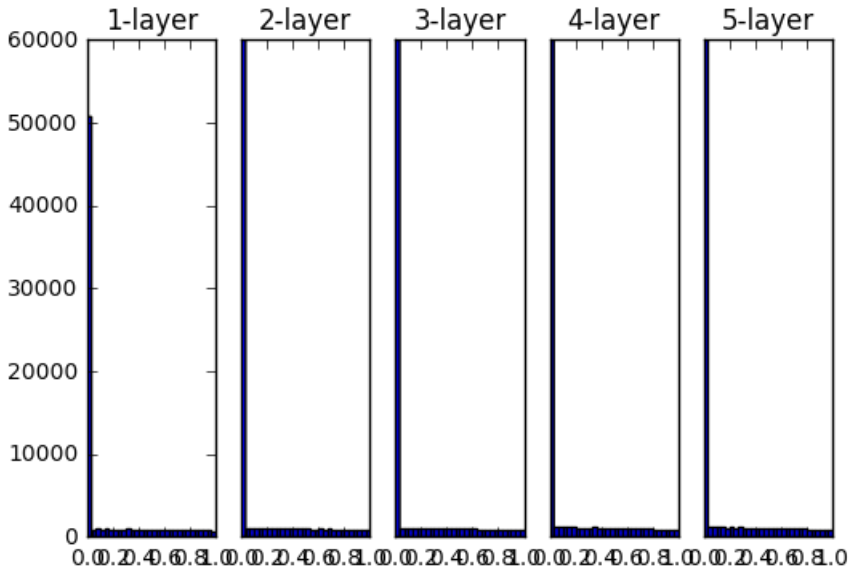
In [15]:
```
# 표준편차가 0.01인 정규분포를 가중치 초기값으로 사용한 경우
w = np.random.randn(node_num, node_num) * 0.01
z = ReLU
activations = get_activation(hidden_layer_size, x, w, z)
get_histogram(activations)
```



In [16]:
```
# Xavier 초기값을 사용한 경우
w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)
activations = get_activation(hidden_layer_size, x, w, z)
get_histogram(activations)
```



In [17]:
```
# He 초기값을 사용한 경우
w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)
activations = get_activation(hidden_layer_size, x, w, z)
get_histogram(activations)
```



std = 0.01일 때

각 층의 활성화 값들이 아주 작은 값들. 역전파의 가중치의 기울기 역시 작아짐. 실제로 학습이 거의 이뤄지지 않음

Xavier 초기값일 때

층이 깊어지면 활성화값들이 치우침. 학습할 때 '기울기 소실'문제.

He 초기값일 때

모든 층에서 균일하게 분포.

실험결과

활성화 함수로 ReLU를 사용할 때 He 초기값.

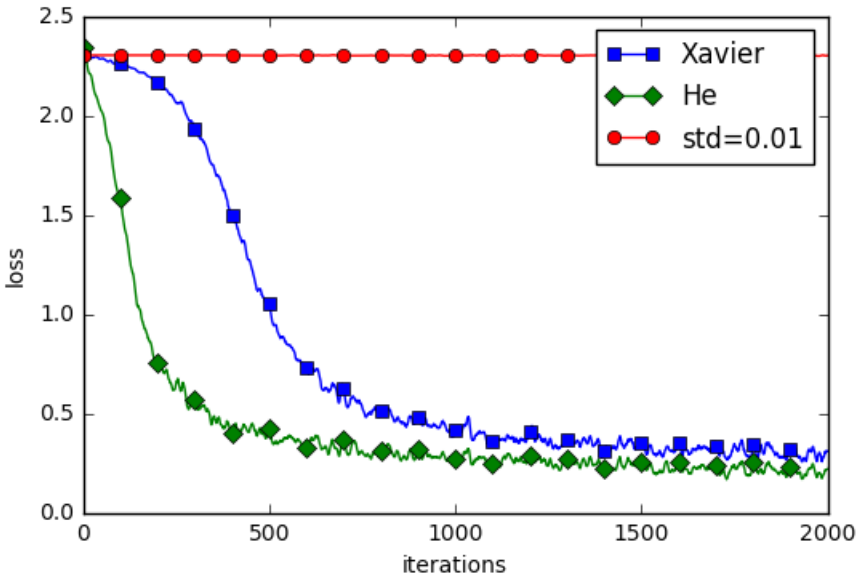
활성화 함수로 sigmoid, tanh 등 S자 모양 곡선일 때는 Xavier 초기값.

6.2.4 MNIST 데이터셋으로 본 가중치 초기값

실제 데이터로 가중치의 초기값을 주는 방법이 신경망 학습에 얼마나 영향을 주는지 그래프

In [18]:
```
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()
```

```
==========iteration:0==========
Xavier:2.31384135187
He:2.34498798533
std=0.01:2.30254975513
==========iteration:100==========
Xavier:2.26558723845
He:1.58078348532
std=0.01:2.30196097913
==========iteration:200==========
Xavier:2.17280837847
He:0.810168717465
std=0.01:2.30293591091
==========iteration:300==========
Xavier:1.94970440178
He:0.597089459925
std=0.01:2.30207993687
==========iteration:400==========
Xavier:1.46563913409
He:0.344485908722
std=0.01:2.30431124013
==========iteration:500==========
Xavier:1.05643911646
He:0.385801222157
std=0.01:2.30082606933
==========iteration:600==========
Xavier:0.817172701852
He:0.427096237727
std=0.01:2.30293235202
==========iteration:700==========
Xavier:0.505855271771
He:0.229300751274
std=0.01:2.30316524884
==========iteration:800==========
Xavier:0.578938102139
He:0.426229083181
std=0.01:2.30451597705
==========iteration:900==========
Xavier:0.441434960777
He:0.21146941566
std=0.01:2.30652560511
==========iteration:1000==========
Xavier:0.343254890766
He:0.24493341527
std=0.01:2.30943181847
==========iteration:1100==========
Xavier:0.341174922321
He:0.25864457152
std=0.01:2.30206144001
==========iteration:1200==========
Xavier:0.327187215679
He:0.273478733415
std=0.01:2.30434366593
==========iteration:1300==========
Xavier:0.358360552311
He:0.274783472434
std=0.01:2.30173895696
==========iteration:1400==========
Xavier:0.272255240241
He:0.201860475426
std=0.01:2.30078041863
==========iteration:1500==========
Xavier:0.317536219063
He:0.253962076292
std=0.01:2.29798110558
==========iteration:1600==========
Xavier:0.372065439537
He:0.23625937496
std=0.01:2.30194863104
==========iteration:1700==========
```



층별 뉴런 수가 100개인 5층 신경망에서 활성화 함수로 ReLu를 사용.

표준편차가 0.01일 때 학습이 전혀 이뤄지지 않음

Xaview, He 초기값은 학습이 순조로움. He 초기값이 학습진도가 빠름.

6.3 배치 정규화

배치 정규화: 각 층의 활성화를 적당히 분포되도록 조정

6.3.1 배치 정규화 알고리즘

배치 정규화가 주목받는 이유

- 학습을 빨리 진행할 수 있다.
- 초기값에 크게 의존하지 않는다.
- 오버피팅을 억제한다.

배치정규화: 학습시 미니배치를 단위로 정규화

데이터 분포가 평균이 0, 분산이 1이 되도록 정규화

식 6.7

$$\mu_B := \frac{1}{m}\sum_{i=1}^{m} x_i$$
$$\sigma_B^2 := \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$
$$x_i := \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

미니배치 B={x1, x2, ..., xm}

m개의 입력 데이터의 집합에 대해 평균과 분산을 구함.

그 입력 데이터를 평균이 0, 분산이 1이 되게 (적절한 분포가 되게) 정규화

배치 정규화 계층마다 정규화된 데이터에 고유한 확대와 이동 변환을 수행

식 6.8

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

두 값은 처음에는 1, 0으로 (원본 그대로) 학습하면서 적합한 값으로 조정

배치 정규화의 계산그래프

https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html (https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html)

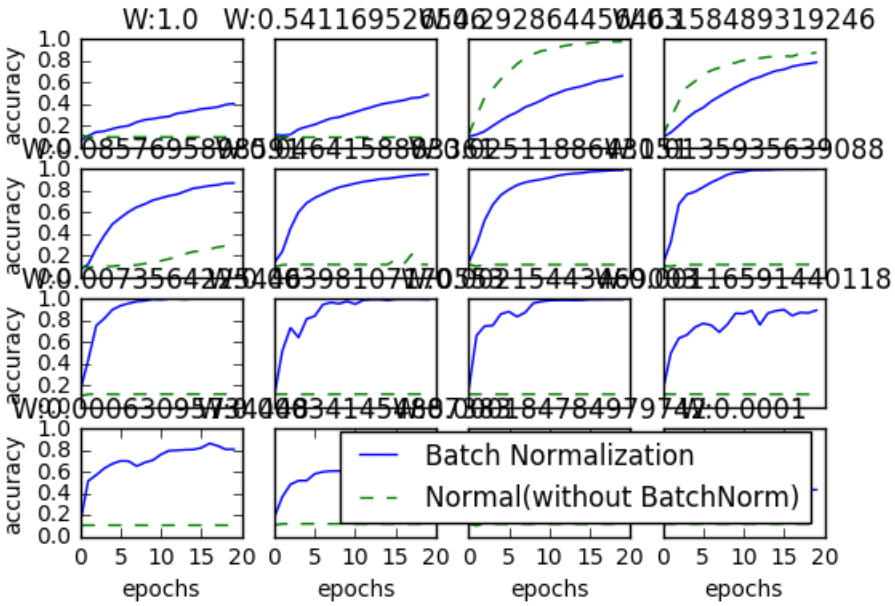
6.3.2 배치 정규화의 효과

MNIST 셋을 사용하여 배치 정규화 계층을 사용할 때와 사용하지 않을 때 학습 진도

가중치 초기값의 표준편차를 다양하게 바꿔가며 학습 경과를 관찰한 그래프

In [19]:
```
# https://github.com/WegraLee/deep-learning-from-scratch/blob/master/ch06/batch_norm_test.py 참고
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.optimizer import SGD, Adam

(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)

# 학습 데이터를 줄임
x_train = x_train[:1000]
t_train = t_train[:1000]

max_epochs = 20
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.01


def __train(weight_init_std):
    bn_network = MultiLayerNetExtend(input_size=784, hidden_size_list=[100, 100, 100, 100, 100], output_size=10,
                                     weight_init_std=weight_init_std, use_batchnorm=True)
    network = MultiLayerNetExtend(input_size=784, hidden_size_list=[100, 100, 100, 100, 100], output_size=10,
                                  weight_init_std=weight_init_std)
    optimizer = SGD(lr=learning_rate)

    train_acc_list = []
    bn_train_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]

        for _network in (bn_network, network):
            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)
            bn_train_acc = bn_network.accuracy(x_train, t_train)
            train_acc_list.append(train_acc)
            bn_train_acc_list.append(bn_train_acc)

            #print("epoch:" + str(epoch_cnt) + " | " + str(train_acc) + " - " + str(bn_train_acc))

            epoch_cnt += 1
            if epoch_cnt >= max_epochs:
                break

    return train_acc_list, bn_train_acc_list


# 그래프 그리기==========
weight_scale_list = np.logspace(0, -4, num=16)
x = np.arange(max_epochs)

for i, w in enumerate(weight_scale_list):
    #print( "============== " + str(i+1) + "/16" + " ==============")
    train_acc_list, bn_train_acc_list = __train(w)

    plt.subplot(4,4,i+1)
```

```
C:\Users\RyanShin\tf\deep-learning\common\functions.py:56: RuntimeWarning: divide by zero encou
ntered in log
  return -np.sum(np.log(y[np.arange(batch_size), t])) / batch_size
C:\Users\RyanShin\tf\deep-learning\common\layers.py:12: RuntimeWarning: invalid value encounter
ed in less_equal
  self.mask = (x <= 0)
C:\Users\RyanShin\tf\deep-learning\common\multi_layer_net_extend.py:100: RuntimeWarning: overfl
ow encountered in square
  weight_decay += 0.5 * self.weight_decay_lambda * np.sum(W**2)
C:\Users\RyanShin\tf\deep-learning\common\multi_layer_net_extend.py:100: RuntimeWarning: invali
d value encountered in double_scalars
  weight_decay += 0.5 * self.weight_decay_lambda * np.sum(W**2)
C:\Users\RyanShin\Anaconda3\lib\site-packages\matplotlib\axes\_axes.py:531: UserWarning: No lab
elled objects found. Use label='...' kwarg on individual plots.
  warnings.warn("No labelled objects found. "
```



거의 모든 경우에 배치 정규화를 사용할 때 학습 진도가 빠른 것으로 나타남

6.4 바른 학습을 위해

오버피팅: 신경망의 훈련 데이터에만 지나치게 적응되어 그 외의 데이터에 제대로 대응하지 못하는 상태

6.4.1 오버피팅

오버피팅이 발생하는 경우

- 매개변수가 많고 표현력이 높은 모델
- 훈련 데이터가 적음

이 두 요건을 일부러 충족하여 오버티핑 발생.

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

각 층의 뉴런은 100개 활성화 함수는 ReLU를 사용.

In [20]:
```
# https://github.com/WegraLee/deep-learning-from-scratch/blob/master/ch06/overfit_weight_decay
.py 참고
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]
```

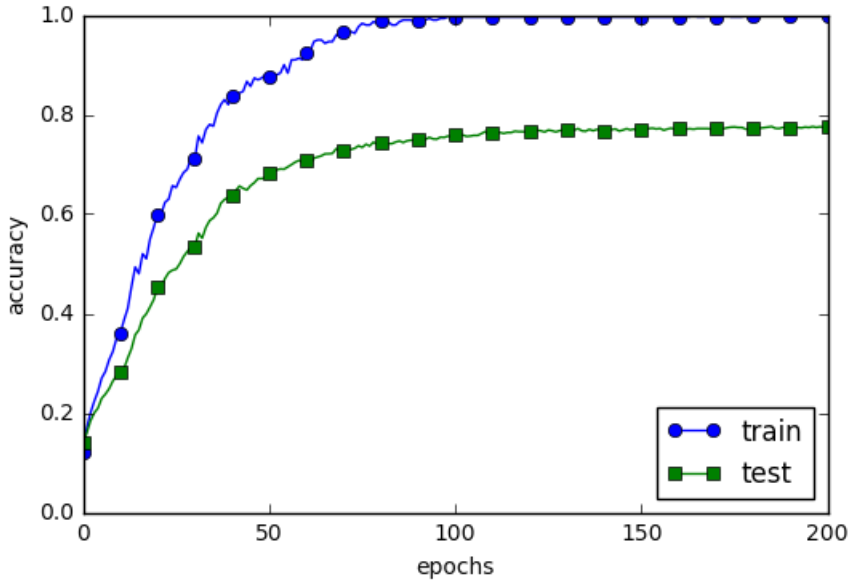
In [21]:
```
import numpy as np
import matplotlib.pyplot as plt
from common.multi_layer_net import MultiLayerNet
#from common.optimizer import SGD

# 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], outpu
t_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:" + st
r(test_acc))

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

```
epoch:0, train acc:0.12, test acc:0.1395
epoch:1, train acc:0.17, test acc:0.1599
epoch:2, train acc:0.2, test acc:0.1841
epoch:3, train acc:0.223333333333, test acc:0.1999
epoch:4, train acc:0.243333333333, test acc:0.2096
epoch:5, train acc:0.27, test acc:0.2292
epoch:6, train acc:0.283333333333, test acc:0.2378
epoch:7, train acc:0.306666666667, test acc:0.2486
epoch:8, train acc:0.323333333333, test acc:0.264
epoch:9, train acc:0.35, test acc:0.2748
epoch:10, train acc:0.36, test acc:0.2808
epoch:11, train acc:0.386666666667, test acc:0.2892
epoch:12, train acc:0.41, test acc:0.3104
epoch:13, train acc:0.453333333333, test acc:0.3288
epoch:14, train acc:0.493333333333, test acc:0.3569
epoch:15, train acc:0.48, test acc:0.367
epoch:16, train acc:0.52, test acc:0.3907
epoch:17, train acc:0.51, test acc:0.3993
epoch:18, train acc:0.55, test acc:0.4132
epoch:19, train acc:0.573333333333, test acc:0.428
epoch:20, train acc:0.6, test acc:0.4524
epoch:21, train acc:0.6, test acc:0.4522
epoch:22, train acc:0.623333333333, test acc:0.4702
epoch:23, train acc:0.63, test acc:0.4821
epoch:24, train acc:0.656666666667, test acc:0.487
epoch:25, train acc:0.653333333333, test acc:0.4892
epoch:26, train acc:0.67, test acc:0.5001
epoch:27, train acc:0.683333333333, test acc:0.5145
epoch:28, train acc:0.69, test acc:0.5244
epoch:29, train acc:0.713333333333, test acc:0.5334
epoch:30, train acc:0.71, test acc:0.5328
epoch:31, train acc:0.756666666667, test acc:0.5615
epoch:32, train acc:0.743333333333, test acc:0.5514
epoch:33, train acc:0.77, test acc:0.5732
epoch:34, train acc:0.78, test acc:0.5867
epoch:35, train acc:0.776666666667, test acc:0.5923
epoch:36, train acc:0.803333333333, test acc:0.6017
epoch:37, train acc:0.82, test acc:0.6215
epoch:38, train acc:0.83, test acc:0.6291
epoch:39, train acc:0.82, test acc:0.6333
epoch:40, train acc:0.836666666667, test acc:0.6379
epoch:41, train acc:0.836666666667, test acc:0.6467
epoch:42, train acc:0.843333333333, test acc:0.6561
epoch:43, train acc:0.846666666667, test acc:0.6516
epoch:44, train acc:0.866666666667, test acc:0.6482
epoch:45, train acc:0.856666666667, test acc:0.6571
epoch:46, train acc:0.873333333333, test acc:0.6629
epoch:47, train acc:0.87, test acc:0.6712
epoch:48, train acc:0.873333333333, test acc:0.671
epoch:49, train acc:0.88, test acc:0.6736
epoch:50, train acc:0.876666666667, test acc:0.6827
epoch:51, train acc:0.88, test acc:0.6844
epoch:52, train acc:0.88, test acc:0.6886
epoch:53, train acc:0.883333333333, test acc:0.6907
epoch:54, train acc:0.9, test acc:0.6901
epoch:55, train acc:0.883333333333, test acc:0.6945
epoch:56, train acc:0.91, test acc:0.6993
epoch:57, train acc:0.91, test acc:0.7043
epoch:58, train acc:0.913333333333, test acc:0.7078
epoch:59, train acc:0.91, test acc:0.7036
epoch:60, train acc:0.923333333333, test acc:0.7083
epoch:61, train acc:0.926666666667, test acc:0.6979
epoch:62, train acc:0.946666666667, test acc:0.7122
epoch:63, train acc:0.95, test acc:0.7144
epoch:64, train acc:0.95, test acc:0.7169
epoch:65, train acc:0.943333333333, test acc:0.7209
epoch:66, train acc:0.946666666667, test acc:0.7211
epoch:67, train acc:0.946666666667, test acc:0.7218
epoch:68, train acc:0.96, test acc:0.7311
```

train_acc_list와 test_acc_list에는 에폭단위의 정확도를 저장.

이 두 리스트를 다음처럼 그래프로 그림

In [22]:
```
# 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()
```



정확도가 크게 벌어지는 것은 훈련 데이터에만 적응(fitting)한 결과

6.4.2 가중치 감소

가중치 감소(weight decay)

학습 과정에서 큰 가중치에 대해서는 그에 상응하는 큰 패널티를 부과하여 오버피팅을 억제하는 방법

가중치를 W라 하면 L2 법칙에 따른 가중치 감소는 1/2 λ (W**2) 가 되고 이 값을 손실함수에 더함

λ(람다)는 정규화의 세기를 조절하는 하이퍼파라미터. 이 값을 크게 설정할 수록 큰 가중치에 대한 패널티가 커짐

L2 법칙

W = (W1, W2, ..., Wn)이 있다면

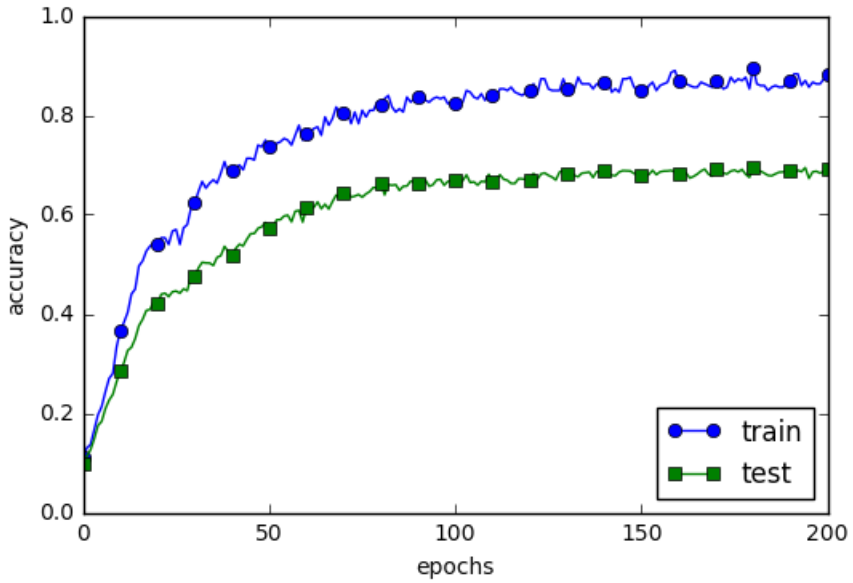
L2 법칙은

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

가중치 감소(λ=0.1)를 적용한 결과

In [23]:
```
# 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], outpu
t_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:" + st
r(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()
```

```
epoch:0, train acc:0.11, test acc:0.0985
epoch:1, train acc:0.13, test acc:0.1132
epoch:2, train acc:0.136666666667, test acc:0.1261
epoch:3, train acc:0.166666666667, test acc:0.1483
epoch:4, train acc:0.196666666667, test acc:0.1752
epoch:5, train acc:0.213333333333, test acc:0.1834
epoch:6, train acc:0.243333333333, test acc:0.2091
epoch:7, train acc:0.27, test acc:0.2263
epoch:8, train acc:0.28, test acc:0.2378
epoch:9, train acc:0.333333333333, test acc:0.2631
epoch:10, train acc:0.366666666667, test acc:0.2851
epoch:11, train acc:0.383333333333, test acc:0.3019
epoch:12, train acc:0.403333333333, test acc:0.3265
epoch:13, train acc:0.44, test acc:0.3327
epoch:14, train acc:0.45, test acc:0.3501
epoch:15, train acc:0.496666666667, test acc:0.3762
epoch:16, train acc:0.506666666667, test acc:0.389
epoch:17, train acc:0.526666666667, test acc:0.4069
epoch:18, train acc:0.536666666667, test acc:0.409
epoch:19, train acc:0.543333333333, test acc:0.4256
epoch:20, train acc:0.54, test acc:0.4201
epoch:21, train acc:0.553333333333, test acc:0.4395
epoch:22, train acc:0.553333333333, test acc:0.4418
epoch:23, train acc:0.54, test acc:0.4347
epoch:24, train acc:0.566666666667, test acc:0.4445
epoch:25, train acc:0.57, test acc:0.4461
epoch:26, train acc:0.54, test acc:0.4417
epoch:27, train acc:0.573333333333, test acc:0.4503
epoch:28, train acc:0.58, test acc:0.4468
epoch:29, train acc:0.613333333333, test acc:0.4752
epoch:30, train acc:0.623333333333, test acc:0.4753
epoch:31, train acc:0.646666666667, test acc:0.4927
epoch:32, train acc:0.666666666667, test acc:0.5038
epoch:33, train acc:0.653333333333, test acc:0.5026
epoch:34, train acc:0.663333333333, test acc:0.5024
epoch:35, train acc:0.67, test acc:0.4976
epoch:36, train acc:0.663333333333, test acc:0.5139
epoch:37, train acc:0.683333333333, test acc:0.5172
epoch:38, train acc:0.706666666667, test acc:0.5361
epoch:39, train acc:0.683333333333, test acc:0.5252
epoch:40, train acc:0.69, test acc:0.5183
epoch:41, train acc:0.693333333333, test acc:0.5317
epoch:42, train acc:0.7, test acc:0.5403
epoch:43, train acc:0.69, test acc:0.5369
epoch:44, train acc:0.713333333333, test acc:0.5502
epoch:45, train acc:0.713333333333, test acc:0.5615
epoch:46, train acc:0.71, test acc:0.5619
epoch:47, train acc:0.74, test acc:0.5724
epoch:48, train acc:0.736666666667, test acc:0.5756
epoch:49, train acc:0.75, test acc:0.584
epoch:50, train acc:0.736666666667, test acc:0.5718
epoch:51, train acc:0.733333333333, test acc:0.5748
epoch:52, train acc:0.743333333333, test acc:0.5894
epoch:53, train acc:0.746666666667, test acc:0.5899
epoch:54, train acc:0.753333333333, test acc:0.5971
epoch:55, train acc:0.75, test acc:0.5988
epoch:56, train acc:0.74, test acc:0.5855
epoch:57, train acc:0.76, test acc:0.6068
epoch:58, train acc:0.78, test acc:0.6085
epoch:59, train acc:0.75, test acc:0.585
epoch:60, train acc:0.763333333333, test acc:0.6133
epoch:61, train acc:0.766666666667, test acc:0.6169
epoch:62, train acc:0.773333333333, test acc:0.6145
epoch:63, train acc:0.776666666667, test acc:0.6256
epoch:64, train acc:0.76, test acc:0.6118
epoch:65, train acc:0.796666666667, test acc:0.6209
epoch:66, train acc:0.78, test acc:0.6112
epoch:67, train acc:0.793333333333, test acc:0.6261
epoch:68, train acc:0.816666666667, test acc:0.6394
```



훈련 데이터에 대한 정확도와 시험 데이터에 대한 정확도는 그림 6-20에 비해 줄었음

오버피팅이 억제됨

6.4.3 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 비율을 곱하여 출력.

드롭아웃 구현

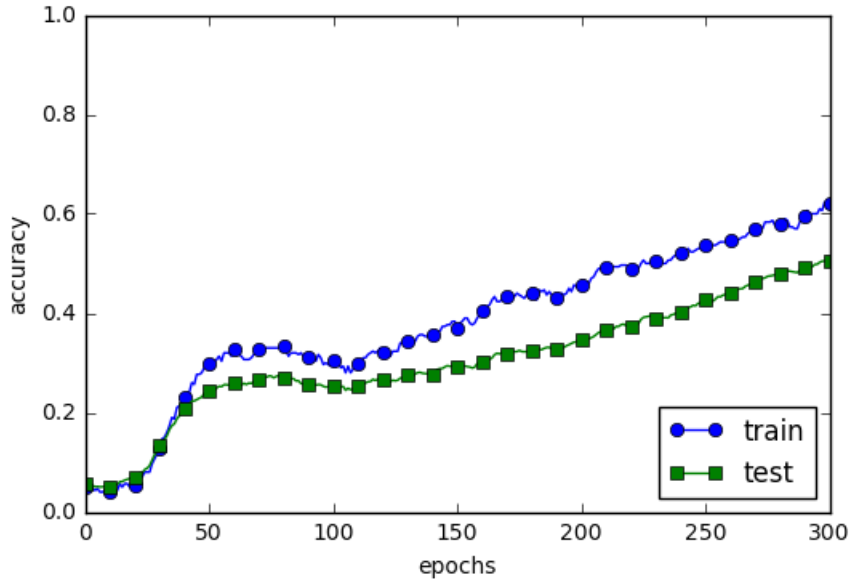
In [24]:
```
class Dropout:
    def __init__(self, dropout_ratio=0.5):
        self.dropout_ratio = dropout_ratio
        self.mask = None

    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.

In [25]:
```
# 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 비울 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=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()
```



드롭아웃을 적용하니 훈련 데이터와 시험 데이터에 대한 정확도 차이가 줄었음

드롭아웃을 이용하면 표현력을 높이면서 오버피팅을 억제가능

앙상블 학습(ensenble learning)

같은 구조의 네트워크를 여러 개 준비하여 따로따로 학습. 여러 개의 출력을 평균내어 답변

드롭아웃은 앙상블 학습과 같은 효과를 (대략) 하나의 네트워크로 구현했다고 볼 수 있음

6.5 적절한 하이퍼파라미터 값 찾기

각 층의 뉴런수, 배치 크기, 매개변수 갱신 시의 학습률과 가중치 감소 등

하이퍼파라미터 값을 최대한 효율적으로 탐색하는 방법

6.5.1 검증 데이터

하이퍼 파라미터 성능을 평가할 때는 시험 데이터를 사용해서는 안됨

- 하이퍼파라미터 값이 시험 데이터에 오버피팅되기 때문

검증 데이터(validation data) : 하이퍼파라미터 전용 확인 데이터

훈련 데이터 : 매개변수 학습

시험 데이터 : 신경망의 범용 성능 평가

데이터셋에 따라서는 훈련 데이터, 검증 데이터, 시험 데이터를 미리 분리해둔 것도 있음

MNIST 데이터셋에서 검증 데이터를 얻는 가장 간단한 방법은 훈련 데이터 중 20% 정도를 검증 데이터로 먼저 분리

In [26]:
```
from dataset.mnist import load_mnist
from common.util import shuffle_dataset

(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:]
```

훈련 데이터를 분리하기 전에 입력 데이터와 정답 레이블을 shuffle_dataset으로 뒤섞음

데이터 셋 안의 데이터가 치우쳐 있을지 모르기 때문

6.5.2 하이퍼파라미터 최적화

하이퍼파라미터 최적화 단계

- 0단계: 하이퍼파라미터 값의 범위를 설정
- 1단계: 설정된 범위에서 하이퍼파라미터의 값을 무작위로 추출

    무작위로 샘플링해 탐색하는 것이 좋은 결과. 최정 정확도에 미치는 영향력이 하이퍼파라미터마다 다르기 때문
    '10의 계승'단위로 범위를 지정. 로그 스케일(log scale)로 지정.

- 2단계: 1단계에서 샘플링한 하이퍼파라미터 값을 사용하여 학습하고, 정확도를 평가. (에폭은 작게 설정)
- 3단계: 1단계와 2단계를 특정 횟수(100회 등) 반복하며, 그 정확도의 결과를 보고 하이퍼파라미터의 범위를 좁힌다.

베이즈 최적화(Bayesian optimization)

베이즈 정리(Bayes' theorem)를 중심으로 한 수학 이론을 구사하여 더 엄밀하고 효율적으로 최적화를 수행

6.5.3 하이퍼파라미터 최적화 구현하기

하이퍼파라미터 검증은 로그 스케일 범위에서 무작위로 추출해 수행

파이썬코드로는 10 ** np.random.uniform(-3,3)처럼 작성.

이 예에서는 가중치 감소 계수를 10^{-8}~10^{-4}, 학습률을 10^{-6}~10^{-2} 범위부터 시작.

하이퍼파라미터 무작위 추출코드

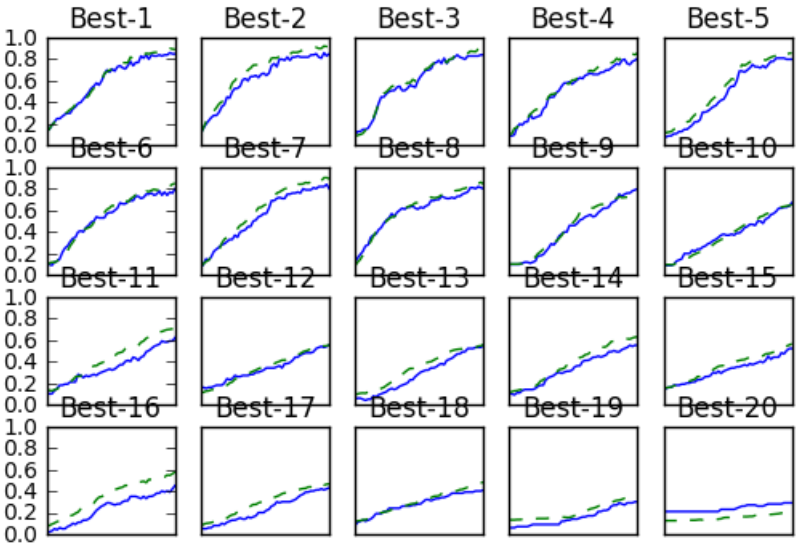
In [27]:
```
weight_decay = 10**np.random.uniform(-8,-4)
lr = 10**np.random.uniform(-6,-2)
```

가중치 감소 계수 범위를 10^{-8} ~ 10^{-4}, 학습률의 범위를 10^{-6}~10^{-2} 실험한 결과

In [28]:
```
# https://github.com/WegraLee/deep-learning-from-scratch/blob/master/ch06/hyperparameter_optim
ization.py 참고
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 = 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(we
ight_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))
```

```
C:\Users\RyanShin\Anaconda3\lib\site-packages\ipykernel\__main__.py:18: VisibleDeprecationWarni
ng: using a non-integer number instead of an integer will result in an error in the future
C:\Users\RyanShin\Anaconda3\lib\site-packages\ipykernel\__main__.py:19: VisibleDeprecationWarni
ng: using a non-integer number instead of an integer will result in an error in the future
C:\Users\RyanShin\Anaconda3\lib\site-packages\ipykernel\__main__.py:20: VisibleDeprecationWarni
ng: using a non-integer number instead of an integer will result in an error in the future
C:\Users\RyanShin\Anaconda3\lib\site-packages\ipykernel\__main__.py:21: VisibleDeprecationWarni
ng: using a non-integer number instead of an integer will result in an error in the future
```

```
val acc:0.1 | lr:1.0342123035963396e-05, weight decay:5.1623482643957004e-05
val acc:0.2 | lr:0.0010324600669850007, weight decay:1.4005992861282985e-05
val acc:0.45 | lr:0.004541693899711486, weight decay:6.625571973850062e-06
val acc:0.1 | lr:0.00022366162386338974, weight decay:2.2363745432704517e-08
val acc:0.1 | lr:3.554423358722337e-05, weight decay:5.440513042802405e-08
val acc:0.14 | lr:8.795835227217176e-05, weight decay:3.5084490693421275e-08
val acc:0.08 | lr:8.389470048723688e-06, weight decay:5.428382486323586e-05
val acc:0.8 | lr:0.008188953103411434, weight decay:1.8234298029421918e-07
val acc:0.12 | lr:4.353141257225612e-06, weight decay:8.544836916069808e-07
val acc:0.15 | lr:3.261132625147742e-05, weight decay:4.16628749080204e-07
val acc:0.85 | lr:0.009971413542256127, weight decay:1.1749861658047927e-08
val acc:0.11 | lr:2.5968224068580865e-05, weight decay:2.243947114476766e-08
val acc:0.07 | lr:9.930380364306747e-05, weight decay:2.1806219065895764e-06
val acc:0.07 | lr:4.005193803755479e-05, weight decay:8.712873214212383e-06
val acc:0.43 | lr:0.0026664033819899257, weight decay:1.1871109433415606e-08
val acc:0.17 | lr:0.0015836514766572143, weight decay:3.157317378237737e-08
val acc:0.04 | lr:0.00016461003228684587, weight decay:4.238749783179235e-06
val acc:0.07 | lr:0.00026261319228131503, weight decay:2.1844706561304078e-06
val acc:0.81 | lr:0.007456009911434539, weight decay:4.51375037490745e-07
val acc:0.07 | lr:8.460237802232236e-05, weight decay:2.9554884633680067e-06
val acc:0.04 | lr:4.888751863721047e-05, weight decay:8.815552568461196e-07
val acc:0.11 | lr:0.0004017624759342226, weight decay:3.7402460874004346e-06
val acc:0.11 | lr:0.0001310823663865221, weight decay:8.599670913886717e-05
val acc:0.1 | lr:7.40122732866785e-06, weight decay:1.5893231077218712e-06
val acc:0.11 | lr:0.0008015702250635178, weight decay:7.917194704491181e-06
val acc:0.8 | lr:0.007278533095822762, weight decay:6.486344214526115e-05
val acc:0.15 | lr:0.00029785266966690177, weight decay:1.1633773834427032e-05
val acc:0.12 | lr:2.9100616267580084e-05, weight decay:9.298069161723102e-05
val acc:0.26 | lr:0.0014798959947988023, weight decay:5.790166787565736e-07
val acc:0.08 | lr:8.081768701746441e-05, weight decay:2.4872439334992756e-06
val acc:0.13 | lr:7.620269817937631e-06, weight decay:1.714324168425928e-06
val acc:0.12 | lr:3.1518488720817987e-06, weight decay:1.6864944374127082e-05
val acc:0.19 | lr:1.366050972243017e-06, weight decay:6.445701072639884e-08
val acc:0.07 | lr:1.5487935414312224e-06, weight decay:5.091726720451668e-07
val acc:0.07 | lr:0.0003875132783809213, weight decay:2.656957752836315e-05
val acc:0.08 | lr:2.7335370707770895e-05, weight decay:1.8736152955903106e-08
val acc:0.12 | lr:4.9981441854336785e-05, weight decay:4.1441520364089146e-08
val acc:0.67 | lr:0.004098474421137249, weight decay:6.201697383475673e-05
val acc:0.12 | lr:6.259266664175609e-05, weight decay:4.5695738014683015e-08
val acc:0.16 | lr:1.4859481525717316e-06, weight decay:1.188057104525221e-05
val acc:0.26 | lr:0.0013871023624257685, weight decay:1.459976150805843e-05
val acc:0.3 | lr:0.001447741021794987, weight decay:1.9539175144117087e-06
val acc:0.14 | lr:9.555832970493743e-05, weight decay:2.983461320367795e-08
val acc:0.23 | lr:0.0007658116840737346, weight decay:1.1003967721793163e-06
val acc:0.06 | lr:3.244867911729493e-06, weight decay:2.4145259380374563e-05
val acc:0.1 | lr:0.0003636003203166488, weight decay:6.262873524428391e-08
val acc:0.24 | lr:0.0013258772444208861, weight decay:3.966657268341709e-05
val acc:0.41 | lr:0.0022692308294192777, weight decay:6.3340410433315666e-08
val acc:0.09 | lr:5.838868207879974e-06, weight decay:5.064631669281342e-07
val acc:0.08 | lr:1.6392358539683672e-05, weight decay:1.8082134349937837e-06
val acc:0.17 | lr:0.00030925531982445336, weight decay:9.40732960723996e-06
val acc:0.55 | lr:0.002556253238476384, weight decay:8.0556435970525e-06
val acc:0.15 | lr:0.00029850517379096406, weight decay:9.825460366360293e-06
val acc:0.84 | lr:0.009075164478759246, weight decay:1.0121823301738507e-07
val acc:0.11 | lr:5.21569311554438e-06, weight decay:2.0645249614361216e-08
val acc:0.16 | lr:0.0001966812642230987, weight decay:3.3939381588049544e-06
val acc:0.08 | lr:6.293361363662192e-05, weight decay:7.063653450563542e-05
val acc:0.09 | lr:2.3066055178318697e-05, weight decay:1.5137642760005445e-08
val acc:0.85 | lr:0.00929671224087357, weight decay:5.107882434621536e-08
val acc:0.13 | lr:0.00012214010913524695, weight decay:1.5340568093391758e-05
val acc:0.12 | lr:1.5016049288303301e-06, weight decay:1.3049376772346623e-08
val acc:0.13 | lr:0.00019486434096478272, weight decay:5.816120270793234e-06
val acc:0.13 | lr:7.26980707465118e-06, weight decay:1.5178952835460695e-05
val acc:0.12 | lr:1.2904687948765016e-05, weight decay:4.8847227705816756e-08
val acc:0.09 | lr:7.730982571080133e-05, weight decay:3.354425563580756e-08
val acc:0.08 | lr:1.047151874463136e-05, weight decay:1.7361977458822166e-08
val acc:0.09 | lr:1.4244596392384843e-06, weight decay:1.408577391906441e-08
val acc:0.13 | lr:2.0653663855435615e-06, weight decay:1.6889271070425825e-07
val acc:0.17 | lr:0.0004543326412344617, weight decay:2.8404269966000064e-06
```



Best-5 정도까지 학습이 순조롭게 진행.

결과로 볼 때 학습률은 0.001 ~ 0.01, 가중치 감소 계수는 10^{-8} ~ 10^{-6} 정도임을 알 수 있음.

다음은 축소된 범위로 똑같은 작업을 반복.

범위를 좁혀가다가 특정 단계에서 최종 하이퍼파라미터 값을 하나 선택.

6.6 정리

이번 장에서 배운 것

매개변수 갱신 방법에는 확률적 경사 하강법(SGD) 외에도 모멘텀, AdaGrad, Adam 등이 있음

가중치 초기값을 정하는 방법은 올바른 학습을 하는데 매우 중요

가중치의 초기값은 Xavier 초기값(Sigmoid, tanh)과 He 초기값(ReLU)이 효과적

배치 정규화(normalization)를 이용하면 학습을 빠르게 진행, 초기값에 영향을 덜 받게 됨

오버피팅을 억제하는 정규화(regularization) 기술로는 가중치 감소와 드롭아웃이 있음

하이퍼파라미터 값 탐색은 최적 값이 존재할 법한 범위를 점차 좁히면서 하는 것이 효과적