UNIT-II

TRAINING TECHNIQUES

Numerical Differentiation – Gradient – Implementing a Training Algorithm - Stochastic Gradient Descent – Momentum – AdaGrad – Adam – Initial Weight Values – Regularization – Validating Hyper parameters. CHAPTER – 4 & 6 (T1)

2.1 Numerical Differentiation

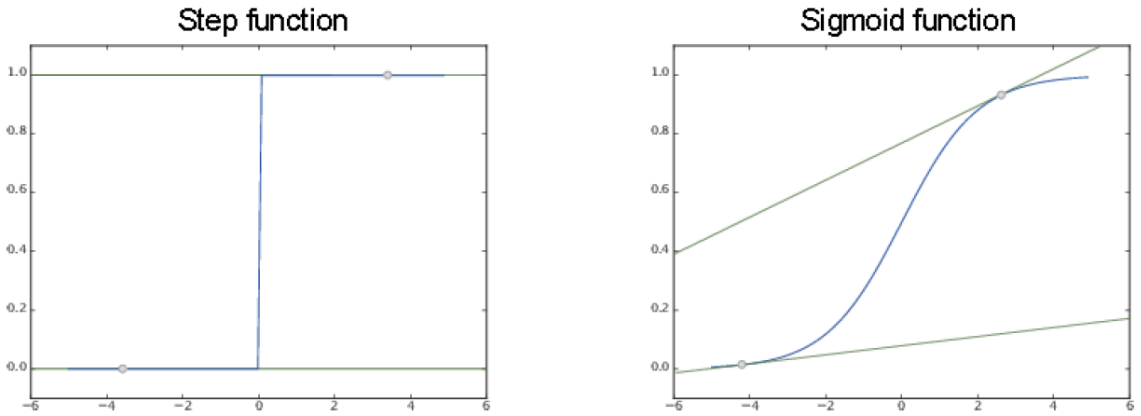


Fig 2.1 Step function and sigmoid function – the gradient of a step function is 0 at almost all positions, while the gradient of a sigmoid function (tangent) is never 0

The gradient method uses information from the gradient to determine which direction to follow.

what a gradient is and its characteristics?

Beginning with a "derivative."

Derivative

A derivative indicates the amount of change at "a certain moment."

Example: 2 km in 10 minutes from the start of a full marathon. You can calculate the speed as *2 / 10 = 0.2* [km/minute]. You ran at a speed of 0.2 km per minute.- "running distance" changed over "time."

Therefore, by minimizing the time of 10 minutes (the distance in the last 1 minute, the distance in the last 1 second, the distance in the last 0.1 seconds, and so on), you can obtain the amount of change at a certain moment (instantaneous speed).

a derivative indicates the amount of change at a certain moment. This is defined by the following equation:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Equation (4.4) indicates the derivative of a function. The left-hand side indicates the derivative of *f(x)* with respect to *x* – the degree of changes of f(x) with respect to *x*. The derivative expressed by equation (4.4) indicates how the value of the function, *f(x)*, changes because of a "slight change" in *x*. Here, the slight change, *h*, is brought close to 0 infinitely, which is indicated as $\lim_{h \to 0}$.

```
# Bad implementation sample
def numerical_diff(f,  x):
    h = 10e-50
    return (f(x+h) - f(x)) / h
```

+

rounding error occurs here

The following example shows a rounding error in Python:

```
>>> np.float32(1e-50)
0.0
```

first improvement. You can use 10−4 as the small value, h. It is known that a value of around 10−4 brings about good results.

The second improvement is in terms of the difference in the function, f. The preceding implementation calculates the difference in the function f between x + h and x. You should observe that this calculation causes an error in the first place.

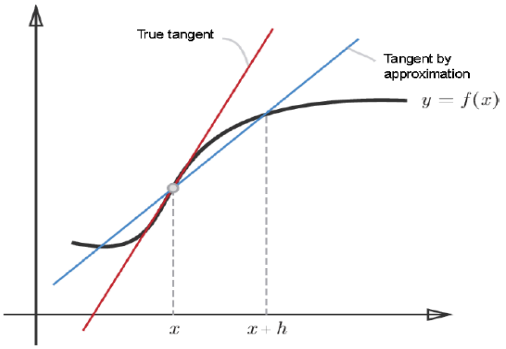


Fig 2.2 True derivative (true tangent) and numerical differentiation (tangent by approximation) are different in value

a numerical differential contains an error. To reduce this error, you can calculate the difference of the function, (*f*), between (*x* + *h*) and (*x* - *h*). This difference is called a **central difference** because it is calculated around *x* (on the other hand, the difference between (*x* + *h*) and *x* is called a **forward difference**). Now, let's implement a numerical differentiation (numerical gradient) based on these two improvements:

```
def numerical_diff(f,  x):
    h = 1e-4 # 0.0001
    return (f(x+h) - f(x-h)) / (2*h)
```

Calculating a derivative by using a very small value difference is called **numerical differentiation**.

Examples of Numerical Differentiation

$$y = 0.01x^2 + 0.1x$$

```
def function_1(x):
    return 0.01*x**2 + 0.1*x
```

The following shows the code for drawing a graph and the resulting graph

```
import numpy as np
import  matplotlib.pylab  as  plt

x = np.arange(0.0, 20.0, 0.1) # The array x containing 0 to 20 in
increments of 0.1
y = function_1(x)
plt.xlabel("x")
plt.ylabel("f(x)")
plt.plot(x, y)
plt.show()
```

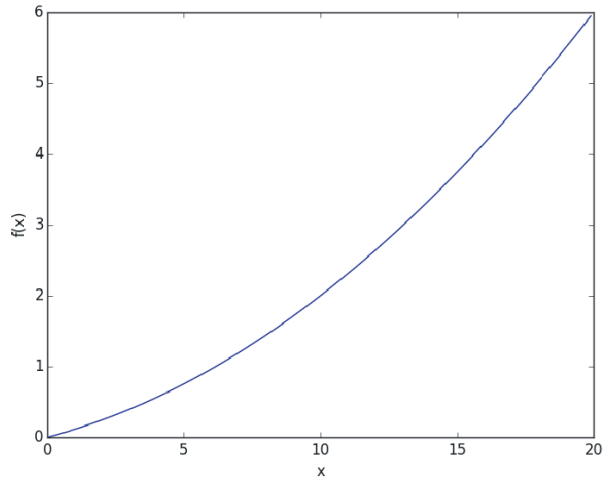


Fig 2. 3 Graph of $f(x) = 0.01x2 + 0.1x$

Now calculate the differentials of the function when x=5 and x=10:

```
>>> numerical_diff(function_1, 5)
0.1999999999990898
>>> numerical_diff(function_1, 10)
0.2999999999986347
```

The differential calculated here is the amount of change of *f(x)* for *x*, which corresponds to the gradient of the function. By the way, the analytical solution of $f(x) = 0.01x2 + 0.1x$ is $= 0.02x + 0.1$. The true derivative when $x=5$ and 10 are 0.2 and 0.3, respectively. They are not strictly identical to the results from numerical differentiation, but the error is very small. Actually, the error is so small that they can be regarded as almost identical values:

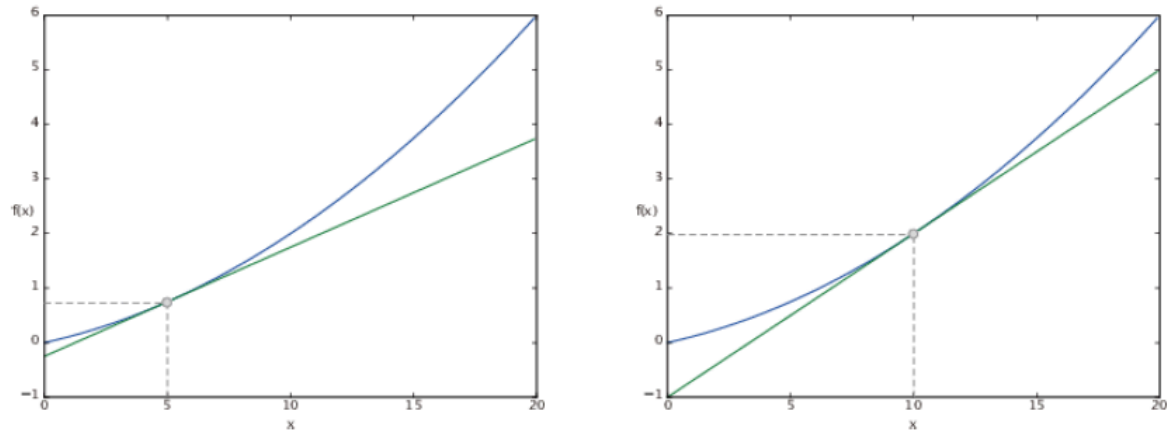


Fig 2.4 Tangents when $x = 5$ and $x = 10$ – using the values from numerical differentiation as the gradients of lines

Partial Derivative

$$f(x_0, x_1) = x_0^2 + x_1^2$$

```
def function_2(x):
    return x[0]**2 + x[1]**2
    # or return np.sum(x**2)
```

To calculate the derivative of equation. Here, please note that equation (4.6) has two variables. Therefore, you must specify for which of the two variables, $x0$ and $x1$, the differentials are calculated. The derivative of a function that consists of multiple variables is called a **partial derivative**.

Question 1: Calculate the partial derivative, for **x0** when **x0 = 3** and **x1 = 4**:

```
>>> def function_tmp1(x0):
... return x0*x0 + 4.0**2.0
...
>>> numerical_diff(function_tmp1, 3.0)
6.00000000000378
```

Question 2: Calculate the partial derivative, for **x1** when **x0 = 3** and **x1 = 4**:

```
>>> def function_tmp2(x1):
... return 3.0**2.0 + x1*x1
...
>>> numerical_diff(function_tmp2, 4.0)
7.999999999999119
```

To solve these problems, a function with one variable is defined, and the derivative for the function is calculated.

For example, in **Question 1**, a new function for **x1=4** is defined, and the function, which has only one variable, **x0**, is passed to the function to calculate a numerical differentiation. Based on the results, the answer to **Question 1** is **6.00000000000378**, and the answer to **Question 2** is **7.999999999999119**. They are mostly the same as the solutions from analytical differentiation.

2.2 Gradient

In the previous example, the partial derivatives of $x0$ and $x1$ were calculated for each variable. Now, we want to calculate the partial derivatives of $x0$ and $x1$ collectively. For example, let's calculate the partial derivatives of ($x0$, $x1$) when **x0 = 3** and **x1 = 4** as $\left(\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}\right)$The vector that collectively indicates the partial differentials of all the variables, such as $\left(\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}\right)$ is called a **gradient**.

```
def numerical_gradient(f,   x):
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x) # Generate an array with the same shape as x

    for idx in range(x.size):
        tmp_val = x[idx]
        # Calculate f(x+h)
        x[idx] = tmp_val + h
        fxh1 = f(x)

        # Calculate f(x-h)
```

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

        grad[idx] = (fxh1 - fxh2) / (2*h)
        x[idx] = tmp_val # Restore the original value

    return grad
```

Note : **np.zeros_like(x)** generates an array that has the same shape as **x** and whose elements are all zero.

The **numerical_gradient(f, x)** function takes the **f (function)** and **x (NumPy array)** arguments and obtains numerical differentiations for each element of the NumPy array, **x**.

Example: obtain the gradients at points (3, 4), (0, 2), and (3, 0):

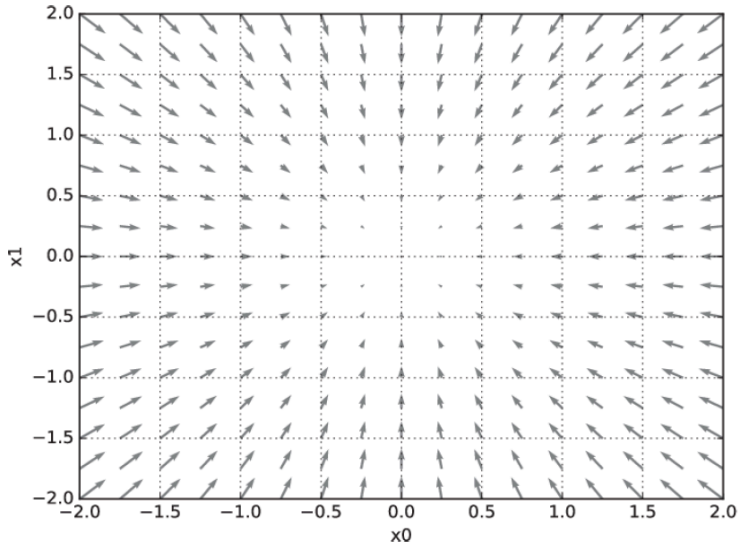
```
>>> numerical_gradient(function_2, np.array([3.0, 4.0]))
array([ 6., 8.])
>>> numerical_gradient(function_2, np.array([0.0, 2.0]))
array([ 0., 4.])
>>> numerical_gradient(function_2, np.array([3.0, 0.0]))
array([  6.,    0.])
```

Note

The actual result is [6.0000000000037801, 7.9999999999991189], but [6., 8.] is returned. This is because a returned NumPy array is formatted to enhance the visibility of the values.

What do these gradients mean? To understand this, let's look at the gradients of . Here, we will make the gradients negative and draw the vectors $f(x_0,x_1) = x_0^2 + x_1^2$

The gradients of $f(x_0,x_1) = x_0^2 + x_1^2$ are shown as the vectors (arrows) that have the direction toward the lowest point.



In the above figure the gradients point at the lowest position, but this is not always the case. In fact, gradient points in the lower direction at each position. To be more precise, the direction of a gradient is **the direction that reduces the value of the function most at each position**.

Gradient Method

Many machine learning problems look for optimal parameters during training. A neural network also needs to find optimal parameters (weights and biases) during training. The optimal parameter here is the parameter value when the loss function takes the minimum value. However, a loss function can be complicated. The parameter space is vast, and we cannot guess where it takes the minimum value. A gradient method makes good use of gradients to find the minimum value (or the smallest possible value) of the function.

A gradient shows the direction that reduces the value of the function most at each position.

Actually, in a complicated function, the direction that a gradient points to is not the minimum value in most cases.

In the gradient method, you move a fixed distance from the current position in the gradient direction. By doing this, you obtain a gradient at the new position and move in the gradient direction again. Thus, you move in the gradient direction repeatedly. Reducing the value of a function gradually by going in the gradient direction repeatedly is known as the **gradient method**. This method is often used in optimization problems for machine learning. It is typically used when training neural networks.

The method for the minimum value is called the **gradient descent method**, while the method for the maximum value is called the **gradient ascent method**.

let's express a gradient method with an equation.

$$x_0 = x_0 - \eta \frac{\partial f}{\partial x_0}$$

$$x_1 = x_1 - \eta \frac{\partial f}{\partial x_1}$$

η adjusts the amount to be updated. This is called a **learning rate** in neural network. A learning rate determines how much needs to be learned and how much to update the parameters.

The above equation is an update equation for one training instance, and the step is repeated. Each step updates the variable values, and the step is repeated several times to reduce the value of the function gradually. This example has two variables, but even when the number of variables is increased, a similar equation—a partial differential value for each variable—is used for updating.

specify the value of the learning rate, such as 0.01 and 0.001, in advance.

Generally, if this value is too large or too small, you cannot reach a "good place." In neural network training, we usually check whether training is successful by changing the value of the learning rate.

Generally, if this value is too large or too small, you cannot reach a "good place." In neural network training, we usually check whether training is successful by changing the value of the learning rate.

```
def gradient_descent(f,  init_x,  lr=0.01,  step_num=100):
    x = init_x

    for i in range(step_num):
        grad = numerical_gradient(f, x)
        x -= lr * grad

    return x
```

The **f** argument is a function to optimize,

the **init_x** argument is an initial value,

the **lr** argument is a learning rate, and

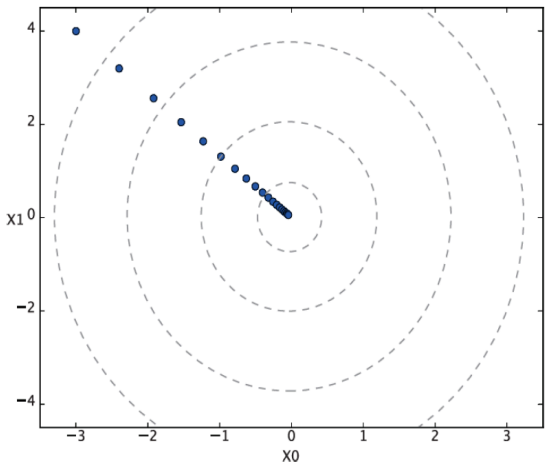
the **step_num** argument is the number of repetitions in a gradient method.

The gradient of the function is obtained by **numerical_gradient(f, x)** and the gradient updated by multiplying it by the learning rate, which is repeated the number of times specified by **step_num**.

Question: Obtain the minimum value for the below function with a gradient method:

$$f(x_0, x_1) = x_0^2 + x_1^2$$

```
>>> def function_2(x):
...     return x[0]**2 + x[1]**2
...
>>> init_x = np.array([-3.0, 4.0])
>>> gradient_descent(function_2, init_x=init_x, lr=0.1, step_num=100)
array([ -6.11110793e-10,  8.14814391e-10])
```



Updating the given function with a gradient method – the dashed lines show the contour lines of the function

an overly large or small learning rate does not achieve good results. Let's do some experiments regarding both cases here:

```
# When the learning rate is too large: lr=10.0
>>> init_x = np.array([-3.0, 4.0])
>>>  gradient_descent(function_2,  init_x=init_x,  lr=10.0,  step_num=100)
array([ -2.58983747e+13, -1.29524862e+12])

# When the learning rate is too small: lr=1e-10
>>> init_x = np.array([-3.0, 4.0])
>>> gradient_descent(function_2, init_x=init_x, lr=1e-10, step_num=100)
array([-2.99999994, 3.99999992])
```

The result diverges to a large value if the learning rate is too large. On the other hand, almost no updates occur if the learning rate is too small. Setting an appropriate learning rate is important.

A parameter such as a learning rate is called a **hyperparameter**. It is different from the parameters (weights and biases) of a neural network in terms of its characteristics. Weight parameters in a neural network can be obtained automatically with training data and a training algorithm, while a hyperparameter must be specified manually. Generally, you must change this hyperparameter to various values to find a value that enables good training.

Gradients for a Neural Network

The gradients here are those of a loss function for weight parameters. For example, let's assume that a neural network has the weight W (2x3 array) only, and the loss function is L. In this case, we can express the gradient as $\frac{\partial L}{\partial W}$. The following equation shows this:

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$$

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{13}} \\ \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{23}} \end{pmatrix}$$

Each element of $\frac{\partial L}{\partial W}$ is the partial derivative for each element.

For example, the element at the first row and column, $\frac{\partial L}{\partial w_{11}}$, indicates how a slight change in w11 changes the loss function, L. What is important here is that the shape of $\frac{\partial L}{\partial W}$ is the same as that of W.

Program that calculates a gradient by taking an easy neural network as

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from common.functions import softmax, cross_entropy_error
from common.gradient import numerical_gradient

class simpleNet:
    def __ init __ (self):
        self.W = np.random.randn(2,3) # Initialize with a Gaussian
distribution

    def predict(self, x):
        return np.dot(x, self.W)

    def loss(self, x, t):
        z = self.predict(x)
        y = softmax(z)
        loss = cross_entropy_error(y, t)

        return  loss
```

Here, the **softmax** and **cross_entropy_error** methods in **common/functions.py** are being used. The **numerical_gradient** method in **common/gradient.py** is also being used. The **simpleNet** class has only one instance variable, which is the weight parameters with a shape of 2x3. It has two methods: one is **predict(x)** for prediction, and the other is **loss(x, t)** for obtaining the value of the loss function. Here, the **x** argument is the input data and the **t** argument is a correct label.

```
>>> net = simpleNet()
>>> print(net.W) # Weight parameters
[[ 0.47355232 0.9977393 0.84668094]
 [ 0.85557411 0.03563661 0.69422093]]
>>>
>>> x = np.array([0.6, 0.9])
```

```
>>> p = net.predict(x)
>>> print(p)
[ 1.05414809 0.63071653 1.1328074]
>>> np.argmax(p) # Index for the maximum value
2
>>>
>>> t = np.array([0, 0, 1]) # Correct label
>>> net.loss(x, t)
0.92806853663411326
```

Next, let's obtain the gradients,using **numerical_gradient(f, x)**. The **f(W)** function defined here takes a dummy argument, **W**. Because the **f(x)** function is executed inside **numerical_gradient(f, x)**, **f(W)** is defined for consistency:

```
>>> def f(W):
...     return net.loss(x, t)
...
>>> dW = numerical_gradient(f, net.W)
>>> print(dW)
[[ 0.21924763 0.14356247 -0.36281009]
 [ 0.32887144 0.2153437 -0.54421514]]
```

The **f** argument of **numerical_gradient(f, x)** is a function and the **x** argument is the argument to the function, **f**. Therefore, a new function, **f**, is defined here. It takes **net.W** as an argument and calculates the loss function. The newly defined function is passed to **numerical_gradient(f, x)**.

numerical_gradient(f, net.W) returns **dW**, which is a two-dimensional 2x3 array. **dW** shows that $\frac{\partial L}{\partial w_{11}}$ for $\frac{\partial L}{\partial W}$ is around **0.2**, for example. This indicates that when *w*11 is increased by h, the value of the loss function increases by 0.2h. $\frac{\partial L}{\partial w_{23}}$ is about **-0.5**, which indicates that when *w*23 is increased by h, the value of the loss function decreases by 0.5h. Therefore, to reduce the loss function, you should update *w*23 in a positive direction and w11 in a negative direction. You can also see that updating w23 contributes to the reduction more than updating *w*11.

use a **lambda** notation to write and implement a simple function

```
>>> f = lambda w: net.loss(x, t)
>>> dW = numerical_gradient(f, net.W)
```

2.3 Implementing a Training Algorithm

A neural network has adaptable weights and biases. Adjusting them so that they fit the training data is called "training." Neural network training consists of four steps.

Step 1 (mini-batch)

Select some data at random from the training data. The selected data is called a mini-batch. The purpose here is to reduce the value of the loss function for the mini-batch.

Step 2 (calculating gradients)

To reduce the loss function for the mini-batch, calculate the gradient for each weight parameter. The gradient shows the direction that reduces the value of the loss function the most.

Step 3 (updating parameters)

Update the weight parameters slightly in the gradient direction.

Step 4 (repeating)

Repeat *steps 1*, *2*, and *3*.

This method uses a gradient descent method to update parameters. Because the data used here is selected at random as a mini-batch, it is referred to as **stochastic gradient descent**. "Stochastic" means "selecting data at random stochastically." Therefore, stochastic gradient descent means "the gradient descent method for randomly selected data." In many deep learning frameworks, stochastic gradient descent is usually implemented as the **SGD** function.

TRY:

A Two-Layer Neural Network as a Class

https://cs231n.github.io/

import sys, os

sys.path.append(os.pardir)

from common.functions import *

```
from common.gradient import numerical_gradient
import numpy as np
class TwoLayerNet:

    def __init__(self, input_size, hidden_size, output_size, weight_init_std=0.01):

        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)

    def predict(self, x):
        W1, W2 = self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']

        a1 = np.dot(x, W1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        y = softmax(a2)

        return y

    # x:
    def loss(self, x, t):
        y = self.predict(x)

        return cross_entropy_error(y, t)

    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        t = np.argmax(t, axis=1)
```

```
    accuracy = np.sum(y == t) / float(x.shape[0])
    return accuracy

# x:
def numerical_gradient(self, x, t):
    loss_W = lambda W: self.loss(x, t)

    grads = {}
    grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
    grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
    grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
    grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

    return grads

def gradient(self, x, t):
    W1, W2 = self.params['W1'], self.params['W2']
    b1, b2 = self.params['b1'], self.params['b2']
    grads = {}

    batch_num = x.shape[0]

    # forward
    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    y = softmax(a2)

    # backward
    dy = (y - t) / batch_num
    grads['W2'] = np.dot(z1.T, dy)
    grads['b2'] = np.sum(dy, axis=0)
```

```
    dz1 = np.dot(dy, W2.T)
    da1 = sigmoid_grad(a1) * dz1
    grads['W1'] = np.dot(x.T, da1)
    grads['b1'] = np.sum(da1, axis=0)

    return grads
```

Variables used in the TwoLayerNet class:

Variable	Description
`params`	Dictionary variable (instance variable) that contains the parameters of the neural network. `params['W1']` is the weights for layer 1, while `params['b1']` is the biases for layer 1. `params['W2']` is the weights for layer 2, while `params['b2']` is the biases for layer 2.
`grads`	Dictionary variable that contains gradients (return value of the `numerical_gradient()` method). `grads['W1']` is the gradients of the weights for layer 1, while `grads['b1']` is the gradients of the biases for layer 1. `grads['W2']` is the gradients of the weights for layer 2, while `grads['b2']` is the gradients of the biases for layer 2.

Methods used in the TwoLayerNet class

Method	Description
`__init__(self, input_size, hidden_size, output_size)`	Initialize. The arguments are the numbers of neurons in the input layer, in the hidden layer, and the output layer in order from left to right.
`predict(self, x)`	Conduct recognition (making predictions). The **x** argument is image data.
`loss(self, x, t)`	Obtain the value of the loss function. The **x** argument is image data, and the **t** argument is the correct label (the same is true for the arguments of the following three methods).
`accuracy(self, x, t)`	Obtain recognition accuracy.
`numerical_gradient(self, x, t)`	Obtain the gradient for the weight parameter.
`gradient(self, x, t)`	Obtain the gradient for the weight parameter. The fast version of the **`numerical_gradient()`** method. This will be implemented in the next chapter.

The **TwoLayerNet** class has two dictionary variables, **params** and **grads**, as instance variables. The **params** variable contains the weight parameters. For example, the weight parameters for layer 1 are stored in **params['W1']** as a NumPy array. You can access the bias for layer 1 using **params['b1']**. Here is an example:

```
net = TwoLayerNet(input_size=784, hidden_size=100, output_size=10)
net.params['W1'].shape # (784, 100)
net.params['b1'].shape # (100,)
net.params['W2'].shape # (100, 10)
net.params['b2'].shape # (10,)
```

the **params** variable contains all the parameters required for this network. The weight parameters contained in the **params** variable are used for predicting (forward processing). You can make a prediction as follows:

```
x = np.random.rand(100, 784) # Dummy input data (for 100 images)
y = net.predict(x)
```

The **grads** variable contains the gradient for each parameter so that it corresponds to the **params** variable. When you calculate gradients by using the **numerical_gradient()** method, gradient information is stored in the **grads** variable, as follows:

```
x = p.random.rand(100, 784) # Dummy input data (for 100 images)
t = np.random.rand(100, 10) # Dummy correct label (for 100 images)

grads = net.numerical_gradient(x, t) # Calculate gradients

grads['W1'].shape # (784, 100)
grads['b1'].shape # (100,)
grads['W2'].shape # (100, 10)
grads['b2'].shape # (10,)
```


```
x = p.random.rand(100, 784) # Dummy input data (for 100 images)
t = np.random.rand(100, 10) # Dummy correct label (for 100 images)

grads = net.numerical_gradient(x, t) # Calculate gradients

grads['W1'].shape # (784, 100)
grads['b1'].shape # (100,)
grads['W2'].shape # (100, 10)
grads['b2'].shape # (10,)
```


implementation of the methods in **TwoLayerNet**

The **__init__** (**self**, **input_size**, **hidden_size**, **output_size**) method is the initialization method of the class (called when **TwoLayerNet** is generated)

The arguments are the numbers of neurons in the input layer, in the hidden layer, and the output layer in order from left to right.

For handwritten digit recognition, a total of 784 input images that are 28x28 in size are provided and 10 classes are returned. Therefore, we specify the **input_ size=784** and **output_size=10** arguments and set an appropriate value for **hidden_size** as the number of hidden layers.

Here, the weights are initialized by using the random numbers based on Gaussian distribution, and the biases are initialized by 0.

The **loss(self, x, t)** method calculates the value of the loss function. It obtains a cross-entropy error based on the result of **predict()** and the correct label.

The remaining **numerical_gradient(self, x, t)** method calculates the gradient of each parameter. It uses numerical differentiation to calculate the gradient for the loss function of each parameter.

Implementing Mini-Batch Training

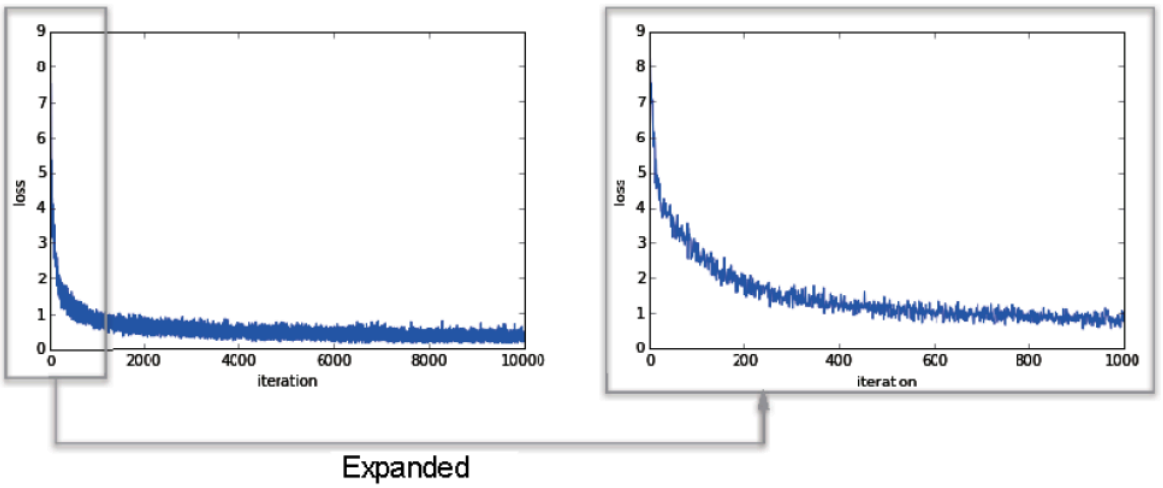
In mini-batch training, we extract some data randomly from training data (called a mini-batch) and use it to update the parameters using a gradient method. Let's conduct training for the **TwoLayerNet** class by using the MNIST dataset

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)
network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)
# Hyper-parameters
iters_num = 10000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

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

iter_per_epoch = max(train_size / batch_size, 1)

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

   # Calculate a gradient
    #grad = network.numerical_gradient(x_batch, t_batch)# fast version!
    grad = network.gradient(x_batch, t_batch)

    # Update the parameters
```

```
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]
    # Record learning progress
    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)

    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("train acc, test acc | " + str(train_acc) + ", " + str(test_acc))
markers = {'train': 'o', 'test': 's'}
x = np.arange(len(train_acc_list))
plt.plot(x, train_acc_list, label='train acc')
plt.plot(x, test_acc_list, label='test acc', linestyle='--')
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```

Here, the size of a mini-batch is 100. Each time, 100 pieces of data (image data and correct label data) are extracted randomly from 60,000 pieces of training data. Then, gradients are obtained for the mini-batch, and the parameters are updated using **stochastic gradient descent** (**SGD**). Here, the number of updates made by a gradient method;that is, the number of iterations is 10,000. At each update, the loss function for the training data is calculated, and the value is added to the array. *Figure* shows the graph of how the value of this loss function changes.

Figure shows that, as the number of training increases, the value of the loss function decreases. It indicates that training is successful. The weight parameters of the neural network are adapting to the data gradually. The neural network is indeed learning. By being exposed to data repeatedly, it is approaching the optimal weight parameters:



Transition of the loss function – the image on the left shows the transition up to 10,000 iterations, while the image on the right shows the transition up to 1,000 iterations

Using Test Data for Evaluation

The reduction in the value of the loss function for the training data indicates that the neural network is learning well. However, this result does not prove that it can handle a different dataset as well as this one.

In neural network training, we must check whether data other than training data can be recognized correctly. We must check whether "overfitting" does not occur. Overfitting means that only the number of images contained in the training data can be recognized correctly, and those that are not contained there cannot be recognized.

The goal of neural network training is to obtain generalization capability. To do that, we must use data that is not contained in the training data to evaluate the generalization capability of the neural network. In the next implementation, we will record the recognition accuracy for the test data and the training data periodically during training. We will record the recognition accuracy for the test data and the training data for each epoch.

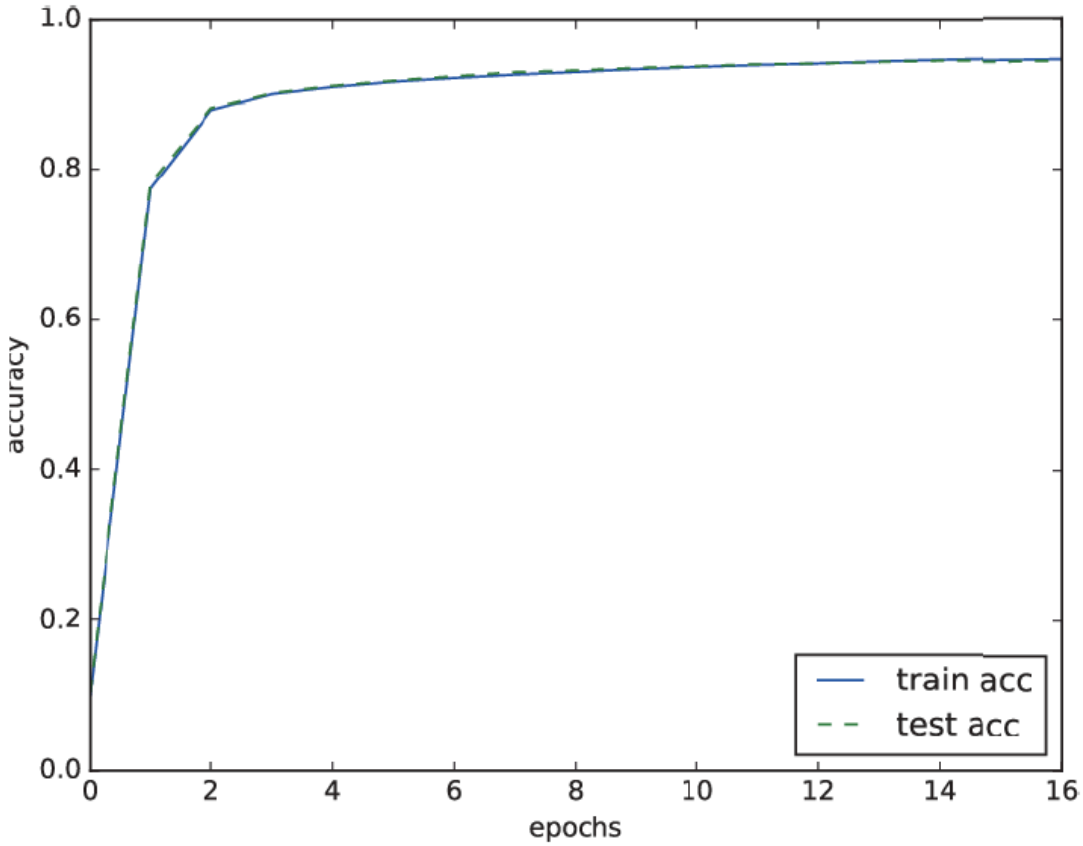
Note

An epoch is a unit. One epoch indicates the number of iterations when all the training data has been used for training. For example, let's assume that 100 mini-batches are used to learn 10,000 pieces of training data. After a stochastic gradient descent method is repeated 100 times, all the training data has been seen. In this case, **100 iterations = 1 epoch**.

change the previous implementation slightly to gain a correct evaluation. Here, the differences from the previous implementation are shown in bold:

```
import numpy as np
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

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

train_loss_list = []
train_acc_list = []
test_acc_list = []
# Number of iterations per epoch
```

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

# Hyper-parameters
iters_num = 10000
batch_size = 100
learning_rate = 0.1
network = TwoLayerNet(input_size=784, hidden_size=50,
output_size=10)

for i in range(iters_num):
    # Obtain a mini-batch
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # Calculate a gradient
    grad = network.numerical_gradient(x_batch, t_batch)
    # grad = network.gradient(x_batch, t_batch) # Quick version!

    # Update the parameters
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)

    # Calculate recognition accuracy for each epoch
    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("train acc, test acc | " + str(train_acc) + " , " +
str(test_acc))
```



Transition of recognition accuracy for training data and test data. The horizontal axis shows the epochs

we can see that the two recognition accuracies are almost the same as the two lines mostly overlap. This indicates that overfitting did not occur here.

Updating Parameters

The purpose of neural network training is to find the parameters that minimize the value of the loss function. The problem is finding the optimal parameters—a process called **optimization**.

2.4 In a deep network, it is more difficult because the number of parameters is huge.

So far, we have depended on the gradients (derivatives) of the parameters to find the optimal parameters. By repeatedly using the gradients of the parameters to update the parameters in the gradient direction, we approach the optimal parameters gradually. This is a simple method called **stochastic gradient descent** (**SGD**), but it is a "smarter" method than searching the parameter space randomly.

2.4.1 SGD

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

Here, the weight parameters to update are W and the gradients of the loss function for W are $\frac{\partial L}{\partial W}$. η is the learning rate.

SGD is a simple method that moves a certain distance in the gradient direction.

```
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]
```

The learning rate is retained as an instance variable. We will also define the **update(params, grads)** method, which is called repeatedly in SGD. The arguments, **params** and **grads**, are dictionary variables (as in the implementation of neural networks so far). Like **params['W1']** and **grads['W1']**, each element stores a weight parameter or a gradient. By using the **SGD** class, you can update the parameters in a neural network as follows (the following code is pseudocode that doesn't run):

```
network = TwoLayerNet(...)
optimizer = SGD()

for i in range(10000):
    ...
    x_batch, t_batch = get_mini_batch(...)  # Mini-batch
    grads = network.gradient(x_batch, t_batch)
    params = network.params
    optimizer.update(params, grads)
    ...
```

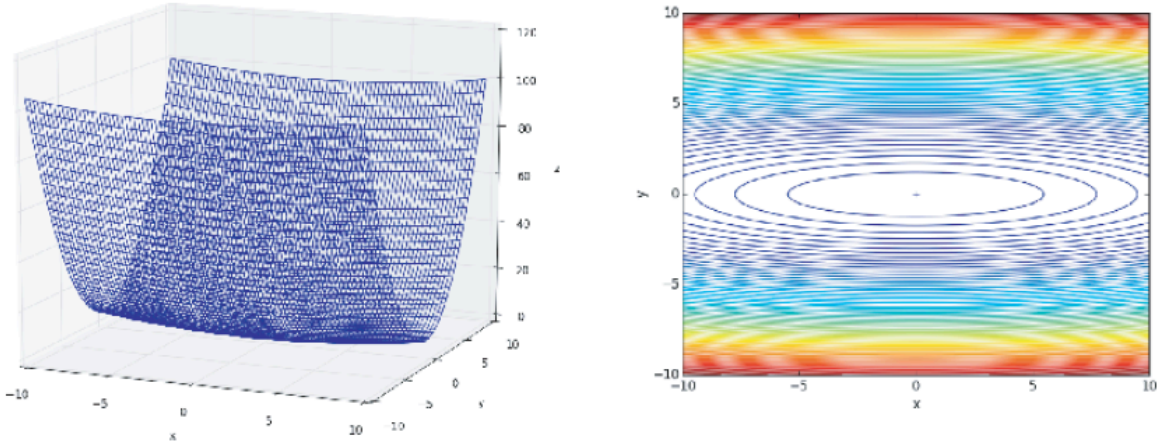
optimizer, means a "person who optimizes." Here, SGD plays this role. The **optimizer** variable takes responsibility for updating the parameters.

Disadvantage of SGD

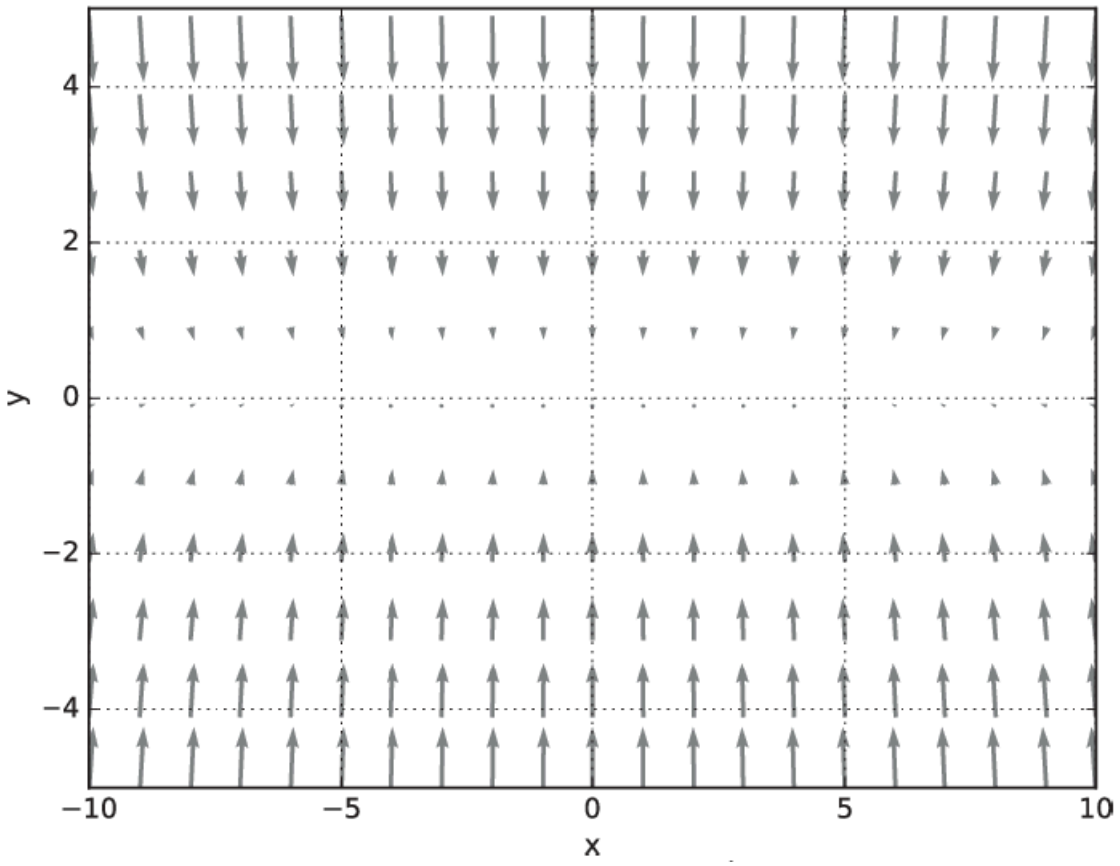
consider a problem that calculates the minimum value of the following function:

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

The shape of the function represented by equation looks like a "bowl" stretched in the *x*-axis direction



Graph of $f(x, y) = \frac{1}{20}x^2 + y^2$ (left) and its contour lines (right)

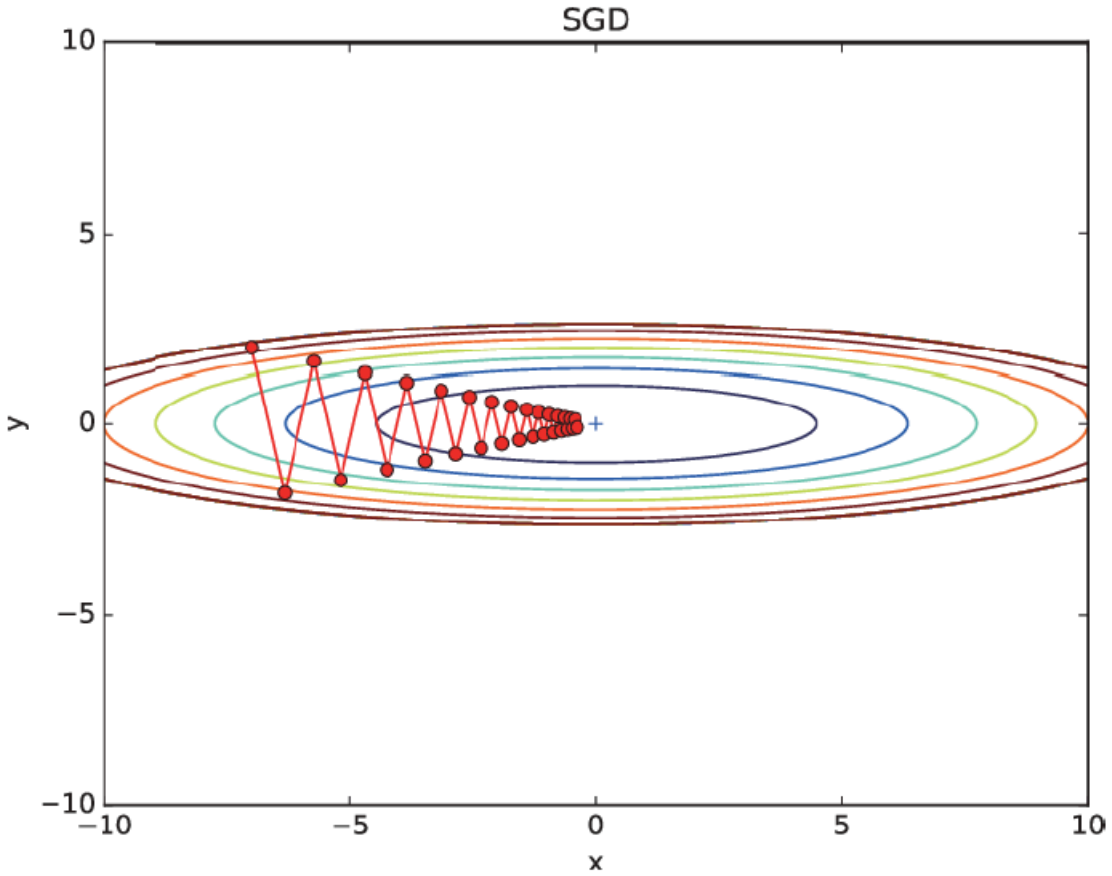


Gradients of $f(x, y) = \frac{1}{20}x^2 + y^2$

These gradients are large in the *y*-axis direction and small in the *x*-axis direction. In other words, the inclination in the *y*-axis direction is steep, while in the *x*-axis direction,

it's gradual. Note that the position of the minimum value of equation is **(x, y) = (0, 0)** but that the gradients in *Figure 6* do not point to the (0, 0) direction in many places.

Let's apply SGD to the function that has the shape shown in the following plots. It starts searching at $(x, y) = (-7.0, 2.0)$ (initial values).



Update path of optimization by SGD – inefficient because it moves in a zigzag to the minimum value (0, 0)

SGD moves in a zigzag. The disadvantage of SGD is that its search path becomes inefficient if the shape of a function is not isotropic—that is, if it is elongated. So, we need a method that is smarter than SGD that moves only in the gradient direction. The root cause of SGD's search path being inefficient is that the gradients do not point to the correct minimum values.

To improve the disadvantage of SGD, we will introduce three alternative methods: Momentum, AdaGrad, and Adam.

2.4.2 Momentum

Momentum is related to physics; it means the "quantity of motion." The Momentum technique is represented by the following equations

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

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

W is the weight parameter to update, $\frac{\partial L}{\partial W}$ is the gradients of the loss function for W, and η is the learning rate.

v, is the "velocity" in physics. Equation represents a physical law stating that an object receives a force in the gradient direction and is accelerated by this force. In Momentum, update functions are used as if a ball had been rolled on the ground, as shown in the following diagram:

Image of Momentum – a ball rolls on the slope of the ground

The term αv in equation slows the object down gradually when it receives no force (a value such as 0.9 is set for α). This is the friction created by the ground or air resistance.

```
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]
```

The instance variable, **v**, retains the velocity of the object. At initialization, **v** retains nothing. When **update()** is called, it retains the data of the same structure as a dictionary variable. The remaining implementation is simple: it just implements equations.

For the equation

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

the update path moves like a ball being rolled around in a bowl. You can see that "the degree of zigzag" is reduced compared to SGD

the update path moves like a ball being rolled around in a bowl. You can see that "the degree of zigzag" is reduced compared to SGD. The force in the *x*-axis direction is very small, but the object always receives the force in the same direction and is accelerated constantly in the same direction. On the other hand, the force in the *y*-axis direction is large, but the object receives the forces in the positive and negative directions alternately. They cancel each other out, so the velocity in the *y*-axis direction is unstable. This can accelerate the motion in the *x*-axis direction and reduce the zigzag motion compared to SGD

2.4.3 AdaGrad

In neural network training, the value of the learning **rate--η** in the **equation--** is important. If it is too small, training takes too long. If it is too large, divergence occurs, and correct training cannot be achieved.

There is an effective technique for the learning rate called **learning rate decay**. It uses a lower learning rate as training advances. This method is often used in neural network training. A neural network learns "much" first and learns "less" gradually.

Reducing the learning rate gradually is the same as reducing the values of the learning rates for all the parameters collectively. AdaGrad is an advanced version of this method. AdaGrad creates a custom-made value for each parameter.

AdaGrad adjusts the learning rate for each element of the parameter adaptively for training (the "Ada" in AdaGrad comes from "Adaptive").

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

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

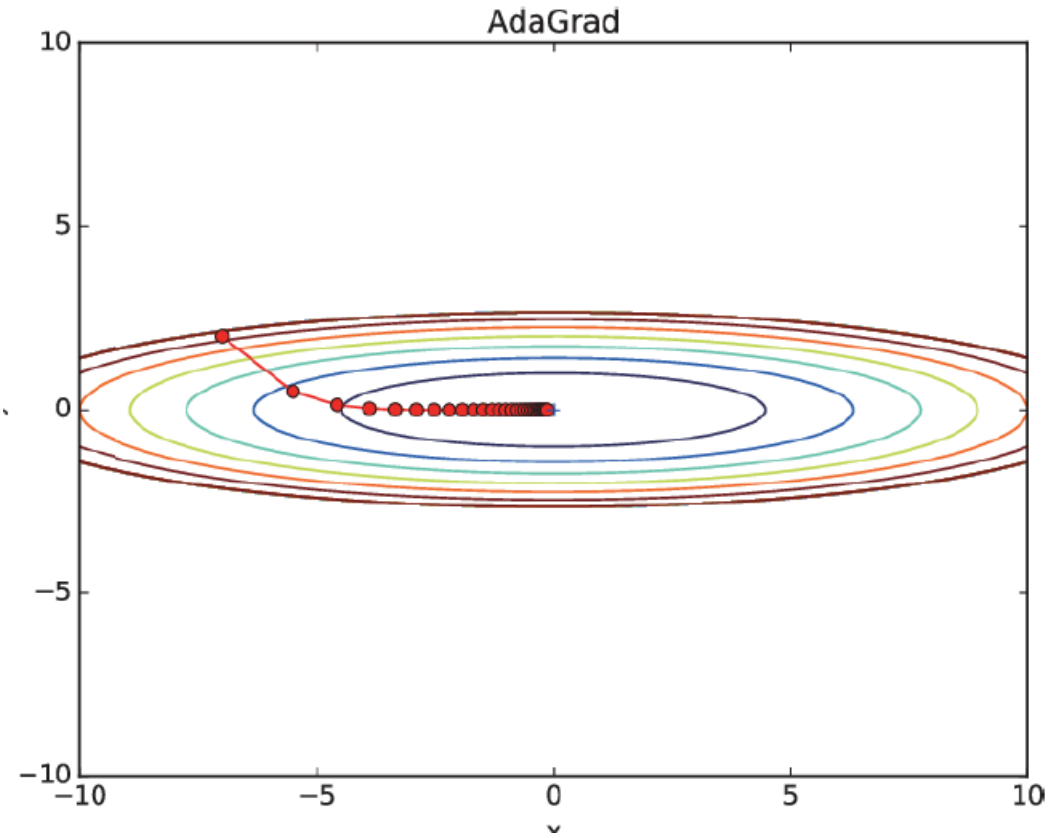
Here, a new variable, h, appears. The h variable stores the sum of the squared gradient values thus far.

When updating parameters, AdaGrad adjusts the scale of learning by multiplying $\frac{1}{\sqrt{h}}$. For the parameter element that moved significantly (i.e., was updated heavily), the learning rate becomes smaller. Thus, you can attenuate the learning rate for each parameter element by gradually reducing the learning rate of the parameter that moved significantly.

```
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)
```

Note that a small value of **1e-7** was added in the last line. This prevents division by **0** when **self.h[key]** contains **0**. In many deep learning frameworks, you can configure this small value as a parameter, but here, a fixed value, **1e-7**, is used.



Update path for optimization by AdaGrad

image shows that the parameters are moving efficiently to the minimum value. The parameters move a lot at first because the gradient in the *y*-axis direction is large. Adjustment is conducted in proportion to the large motion so that the update step becomes small. Thus, the degree of update in the *y*-axis direction is weakened, reducing the zigzag motion.

2.4.4 Adam

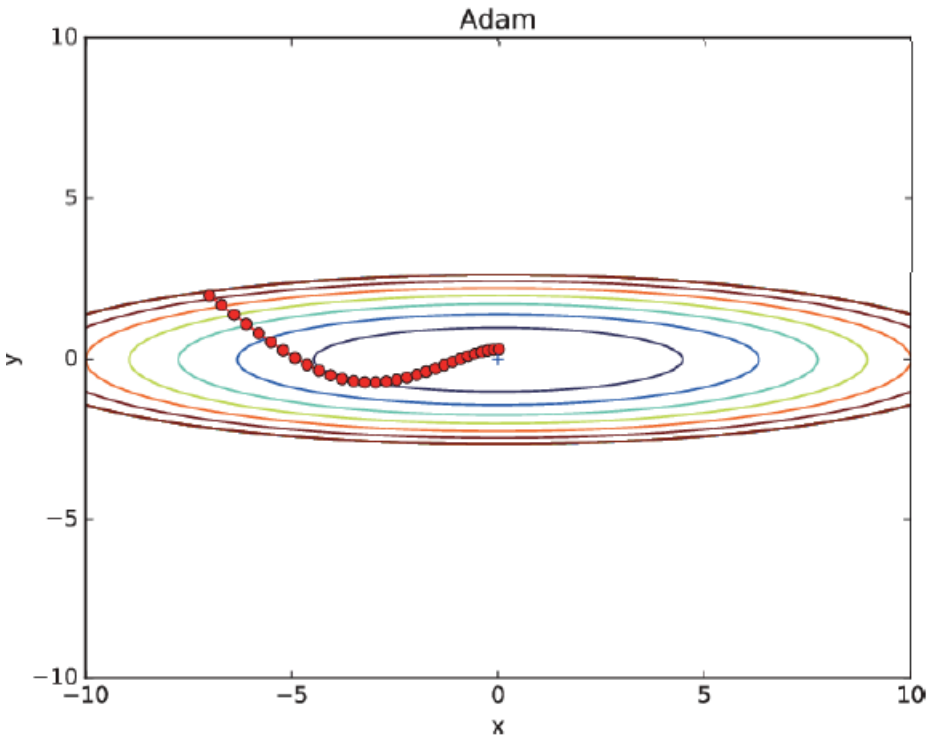
In Momentum, the parameters move based on physical law, such as a ball rolled in a bowl. AdaGrad adjusts the update step adaptively for each parameter element. So, what happens when the two techniques, Momentum and AdaGrad, are combined? This is the basic idea of the technique called Adam

it is like a combination of Momentum and AdaGrad. By combining the advantages of these two techniques, we can expect to search the parameter space efficiently. The "bias correction" of hyperparameters is also a characteristic of Adam.

class Adam:

"""Adam)"""

```
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 bias
        #params[key] += self.lr * unbias_m / (np.sqrt(unbisa_b) + 1e-7)
```

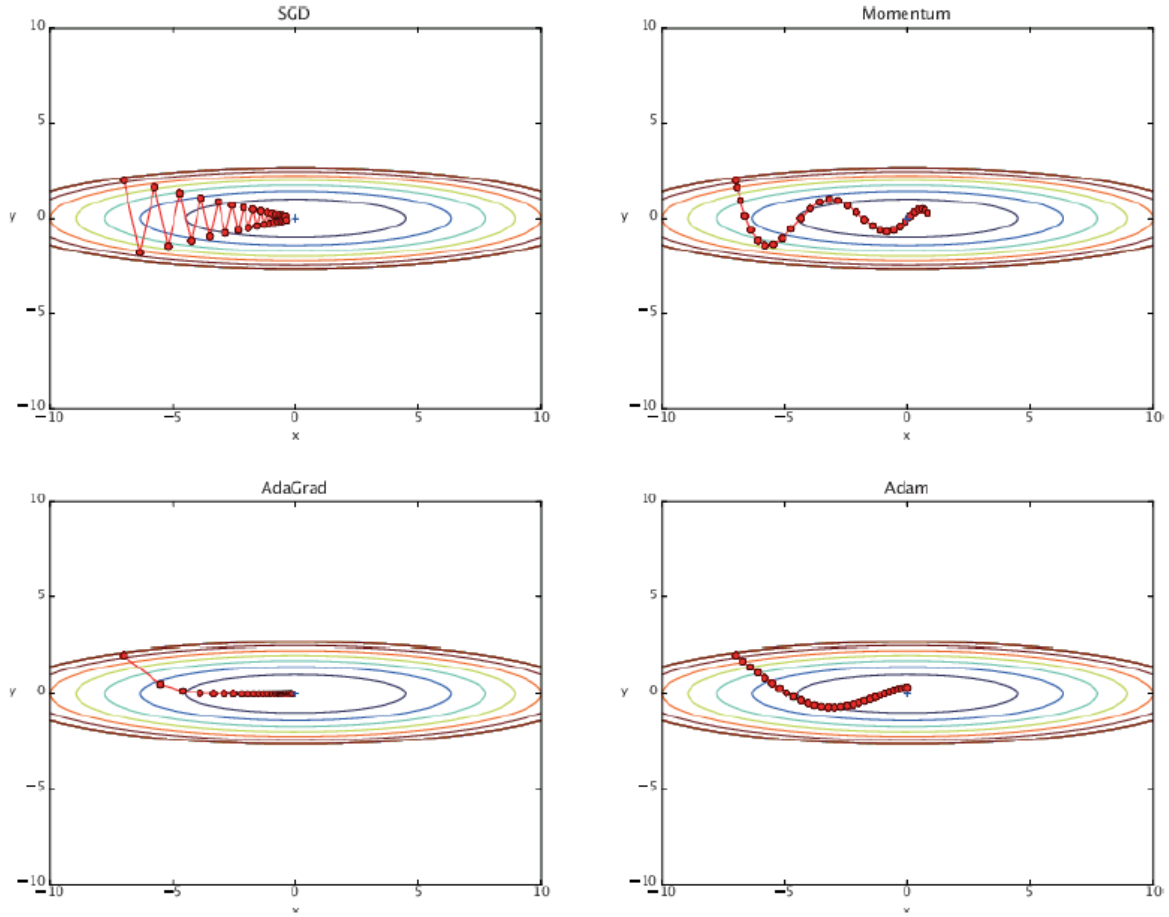


Update path for optimization by Adam

the update path by Adam moves as if a ball has been rolled in a bowl. The motion is similar to that in Momentum, but the left and right motions of the ball are smaller. This advantage is caused by the adaptive adjustment of the learning rate.

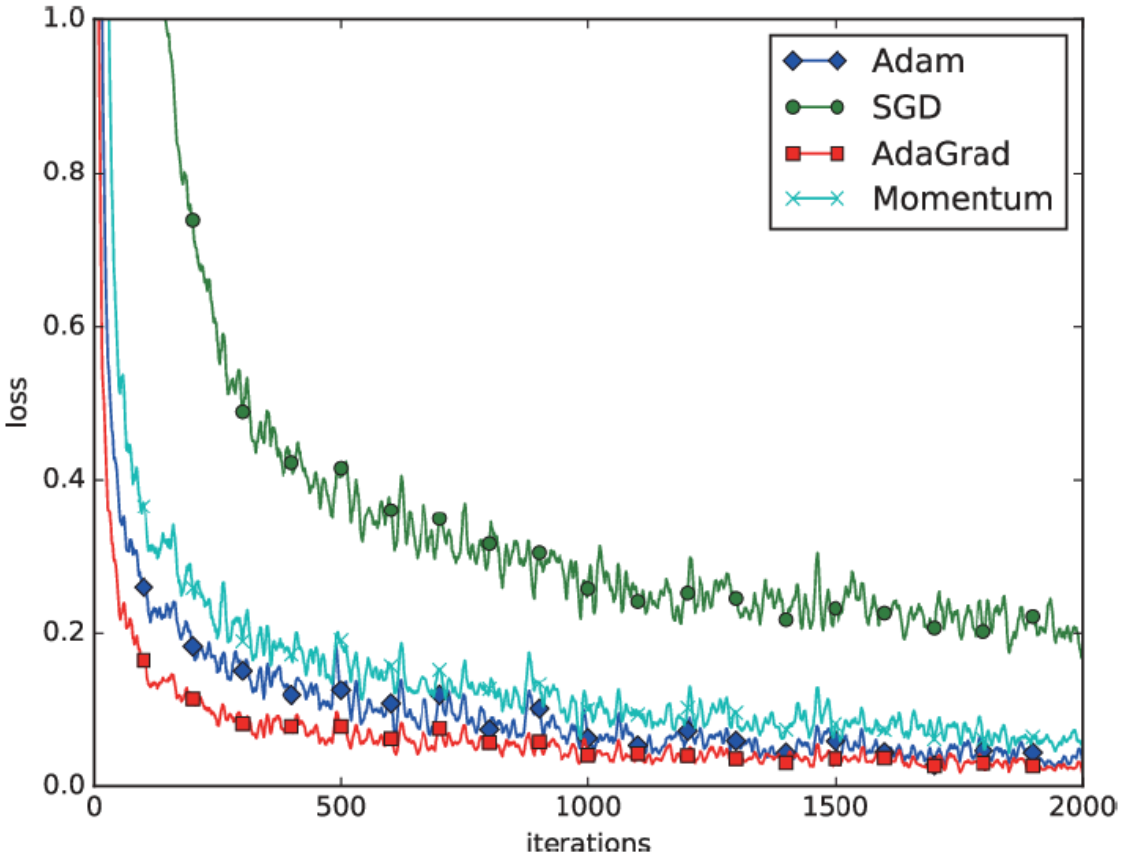
Adam has three hyperparameters. The first is the learning rate (appearing as α in the paper). The others are the coefficient for the primary moment, β1, and the coefficient for the secondary moment, β2. The article states that the standard values are 0.9 for β1 and 0.999 for β2, which are effective in many cases.

Which Update Technique Should We Use?



there is no one technique currently known that is good at solving all problems. Each has its own distinct characteristics and advantages, which make it better suited to certain problems over others. Therefore, it's important to know which technique works best given a specific set of circumstances.

Using the MNIST dataset to compare the four update techniques – the horizontal axis indicates the iterations of learning, while the vertical axis indicates the values of the loss function



2.5 Initial Weight Values

What values are set as the initial weight values often determines the success or failure of neural network training.

How About Setting the Initial Weight Values to 0?

weight decay is a technique that reduces the values of the weight parameters to prevent overfitting.

If we want the weights to be small, starting with the smallest possible initial values is probably a good approach. Here, we use an initial weight value such as **0.01 * np.random.randn(10, 100)**. This small value is the value generated from the Gaussian distribution multiplied by 0.01—a Gaussian distribution with a standard deviation of 0.01.

If we want the weight values to be small, how about setting all the initial weight values to 0? This is a bad idea as it prevents us from training correctly.

Why should the initial weight values not be 0? Or in other words, why should the weights not be uniform values? Well, because all weight values are updated uniformly

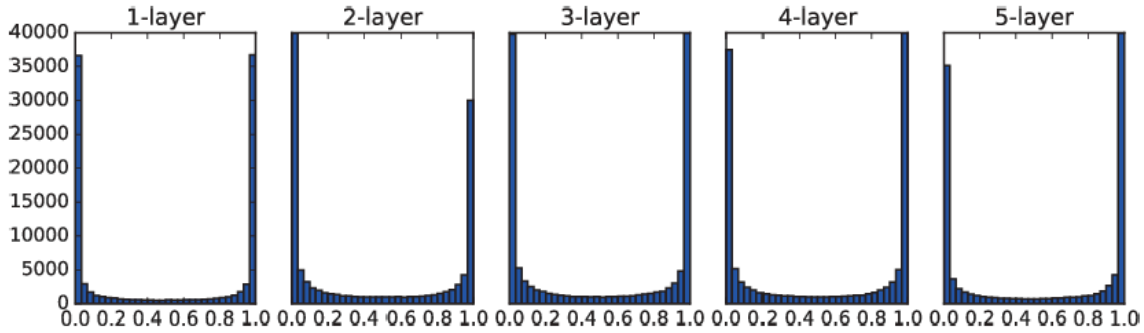
(in the same way) in backpropagation. So, say that layers 1 and 2 have 0 as their weights in a two-layer neural network. Then, in forward propagation, the same value is propagated to all the neurons in layer 2 because the weight of the input layer is 0. When the same values are entered for all the neurons in layer 2, all the weights in layer 2 are updated similarly in backward propagation (please remember "backward propagation in a multiplication node"). Therefore, the weights are updated with the same value and become symmetrical values (duplicate values). Due to this, there is no meaning in having many weights. To prevent the weights from being uniform or breaking their symmetrical structure, random initial values are required.

Distribution of Activations in the Hidden Layers

```
import numpy as np
import matplotlib.pyplot as plt

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

x = np.random.randn(1000, 100) # 1000 data

node_num = 100 # Number of nodes (neurons) in each hidden layer
hidden_layer_size = 5 # Five hidden layers exist
activations = {} # The results of activations are stored here


for i in range(hidden_layer_size):
    if i != 0:
        x = activations[i-1]

    w = np.random.randn(node_num, node_num) * 1

    z = np.dot(x, w)
    a = sigmoid(z) # Sigmoid function!
    activations[i] = a
```

Here, there are five layers and that each layer has 100 neurons. As input data, 1,000 pieces of data are generated at random with Gaussian distribution and are provided to the five-layer neural network. A sigmoid function is used as the activation function, and the activation results of each layer are stored in the **activations** variable. Please note the weight scale. Here, a Gaussian distribution with a standard deviation of 1 is being used. The purpose of this experiment is to observe how the distribution of **activations** changes by changing this scale (standard deviation). Now, let's show the data of each layer that is stored in **activations** in a histogram:

```
# Draw histograms
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()
```

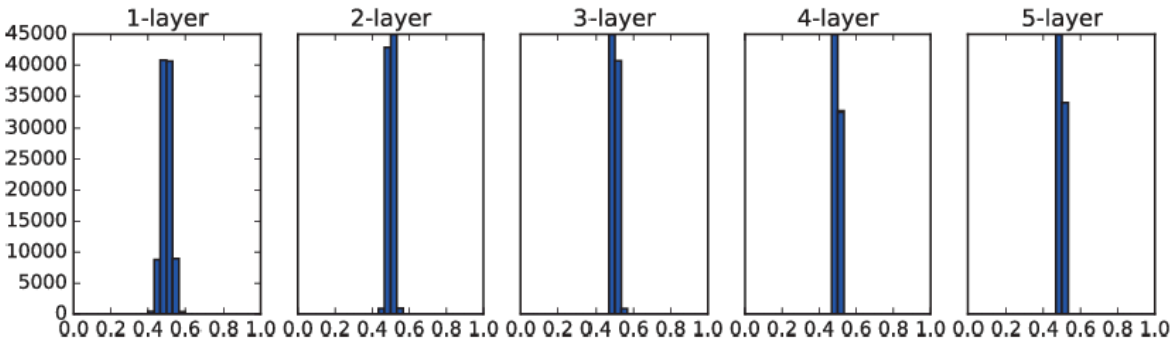


Distribution of the activations of each layer when a Gaussian distribution with a standard deviation of 1 is used for the initial weight values

This image shows that the activations of each layer are mainly 0 and 1. The sigmoid function that's being used here is an S-curve function. As the output of the sigmoid function approaches 0 (or 1), the value of the differential approaches 0. Therefore, when the data is mainly 0s and 1s, the values of the gradients in backward propagation get smaller until they vanish. This is a problem called **gradient vanishing**. In deep learning, where there's a large number of layers, gradient vanishing can be a more serious problem.

let's conduct the same experiment, but this time with the standard deviation of the weights as 0.01. To set the initial weight values, you will need to modify the previous code, as follows:

```
# w = np.random.randn(node_num, node_num) * 1
w = np.random.randn(node_num, node_num) * 0.01
```



Distribution of the activations of each layer when a Gaussian distribution with a standard deviation of 0.01 is used for the initial weight values

Now, the activations concentrate around 0.5. Unlike the previous example, they are not biased toward 0 and 1. The problem of gradient vanishing does not occur. However, when activations are biased, it causes a large problem in terms of its representation. If multiple neurons output almost the same values, there is no meaning in the existence of multiple neurons. For example, when 100 neurons output almost the same values, one neuron can represent almost the same thing. Therefore, the biased activations cause a problem because representation is limited.

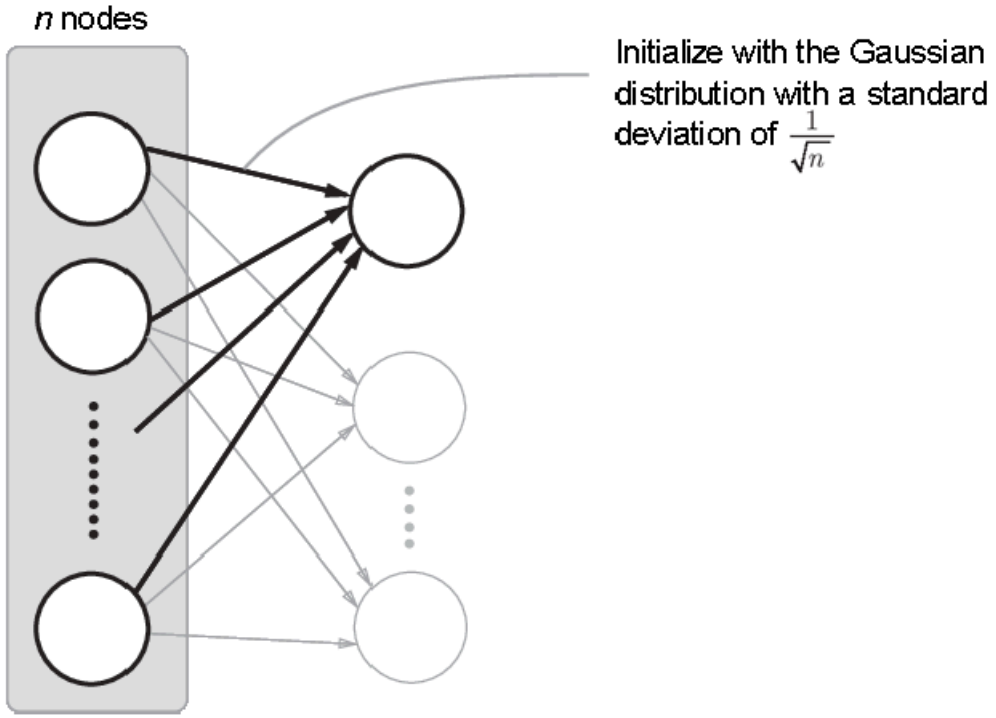
Note

The distribution of the activations in each layer needs to be spread properly. This is because, when moderately diverse data flows in each layer, a neural network learns efficiently. On the other hand, when biased data flows, training may not go well because of the gradient vanishing and "limited representation."

Next, we will use the initial weight values that were recommended in a paper by Xavier Glorot et al.

This is called "Xavier initialization." Currently, the Xavier initializer is usually used in ordinary deep learning frameworks. For example, in the Caffe framework, you can specify the **xavier** argument for the initial weight setting to use the Xavier initializer.

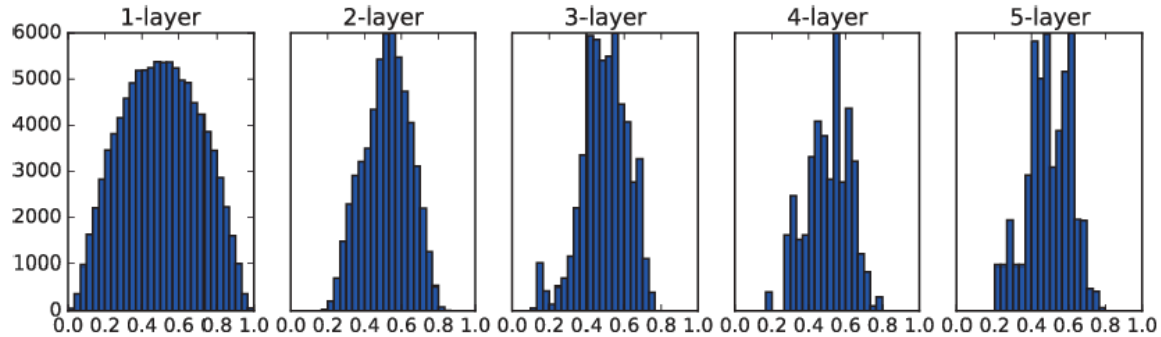
Xavier's paper obtained the appropriate scale of weights so that the activation of each layer was spread similarly. It concluded that distribution with a standard deviation of $\frac{1}{\sqrt{n}}$ should be used when the number of nodes in the previous layer is *n*



Xavier initializer – when *n* nodes in the previous layer are connected, a distribution with the standard deviation of $\frac{1}{\sqrt{n}}$ is used for initial values

When the Xavier initializer is used, since the number of nodes in the previous layer is larger, the weight scale that is set for the initial values for the target nodes is smaller.

```
node_num = 100 # Number of nodes in the previous layer
w = np.random.randn(node_num, node_num) / np.sqrt(node_num)
```



Distribution of the activations of each layer when the Xavier initializer is used as the initial weight value

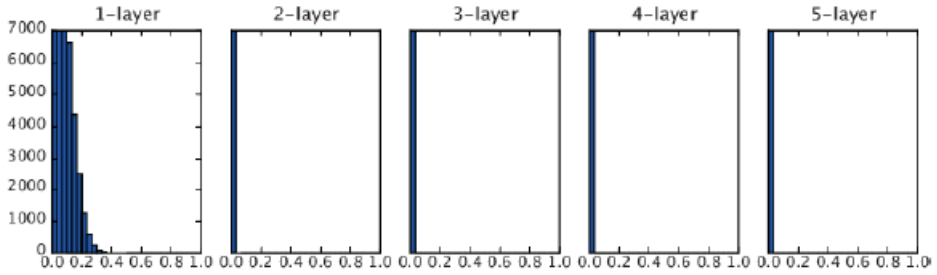
It shows that distributions are spread more widely, although a higher layer has a more distorted shape. We can expect that training is conducted efficiently because the data that flows in each layer is spread properly, and the representation of the sigmoid function is not limited.

Initial Weight Values for ReLU

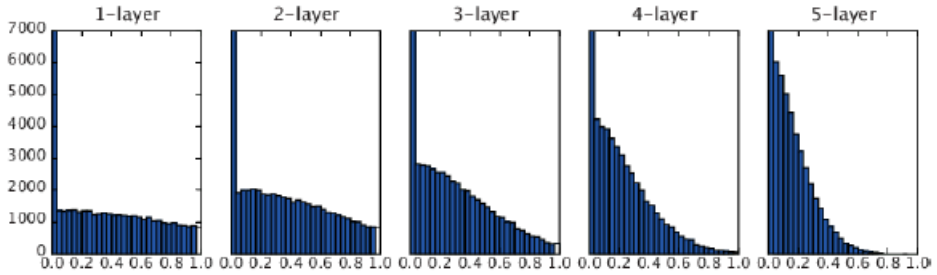
The Xavier initializer is based on the assumption that the activation function is linear. The Xavier initializer is suitable because the **sigmoid** and **tanh** functions are symmetrical and can be regarded as linear functions around their centers. Meanwhile, for ReLU, using the initial value is recommended. This is known as the He initializer and was recommended by Kaiming He and et. Al

The He initializer uses a Gaussian distribution with a standard deviation $\sqrt{\frac{2}{n}}$ of when the number of nodes in the previous layer is *n*. When we consider that the Xavier initializer is $\frac{1}{\sqrt{n}}$, we can assume (intuitively) that the coefficient must be doubled to provide more spread because a negative area is 0 for ReLU.

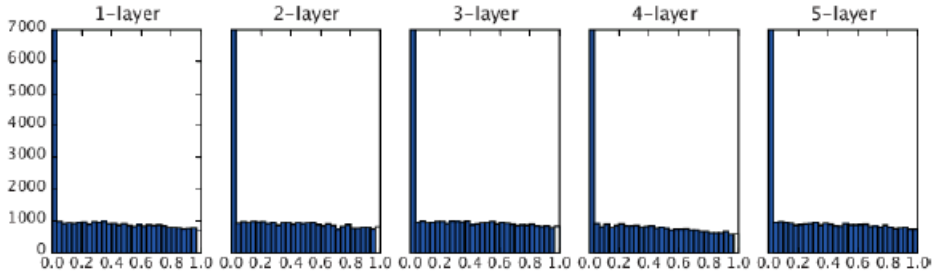
Let's look at the distribution of activations when ReLU is used as the activation function. We will consider the results of three experiments after using a Gaussian distribution with a standard deviation of 0.01 (that is, **std=0.01**), the Xavier initializer, and the He initializer, which is specifically used for ReLU



When the Gaussian distribution with the standard deviation of 0.01 is used for the initial weight values



When the Xavier initializer is used



When the He initializer is used

Change of activation distribution by weight initializers when ReLU is used as the activation function

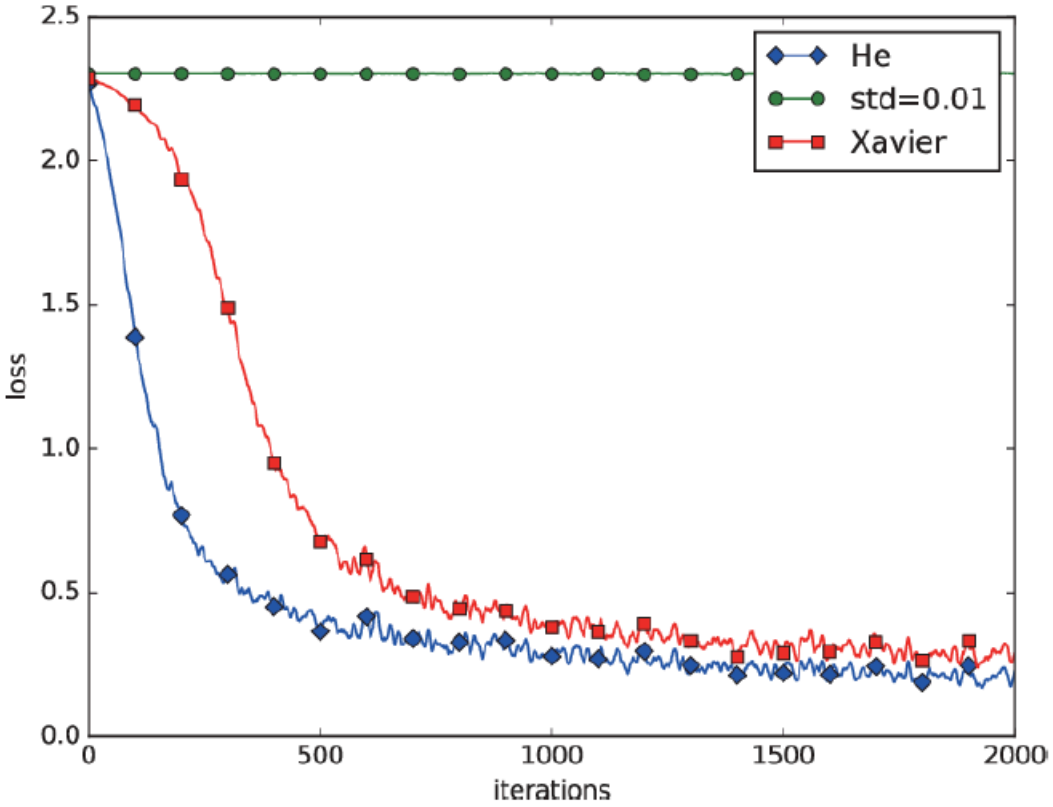
Activations of each layer are very small (the averages of the distributions are as follows: layer 1: 0.0396, layer 2: 0.00290, layer 3: 0.000197, layer 4: 1.32e-5, and layer 5: 9.46e-7) for **std=0.01**.

When small data flows through a neural network, the gradients of the weights in backward propagation are also small. This is a serious problem as training will barely advance.

The results from using the Xavier initializer. This shows that the bias becomes larger little by little as the layers become deeper—as do the activations. Gradient vanishing will be a problem when it comes to training. On the other hand, for the He initializer, the spread of Gaussian distribution in each layer is similar. The spread of data is similar even when the layers are deeper. So, we can expect that appropriate values also flow for backward propagation.

Summary, when you use relu as the activation function, use the He initializer, and for S-curve functions such as **sigmoid** and **tanh**, use the Xavier initializer. As of the time of writing, this is the best practice.

Using the MNIST Dataset to Compare the Weight Initializers



Using the MNIST dataset to compare the weight initializers – the horizontal axis indicates the iterations of training, while the vertical axis indicates the values of the loss function

the initial weight values are very important in neural network training. They often determine their success or failure. Although the importance of the initial weight values is sometimes overlooked, the starting (initial) value is important for everything.

Batch Normalization

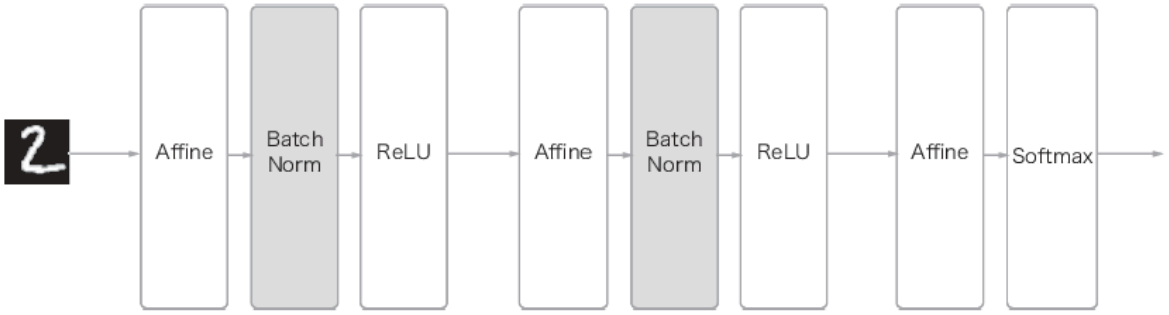
How about adjusting the distribution of activations "forcefully" so that there's a proper spread in each layer? This technique is based on the idea of batch normalization

Batch normalization algorithm

Batch norm attracts a lot of attention due to the following advantages:

- it can accelerate learning (it can increase the learning rate).
- it is not as dependent on the initial weight values (you do not need to be cautious about the initial values).
- it reduces overfitting (it reduces the necessity of dropout).

The first advantage is particularly attractive because deep learning takes a lot of time. With batch norm there's no need to be anxious about the initial weight values, and due to it reducing overfitting, it removes this cause of anxiety from deep learning.
The purpose of batch norm is to adjust the distribution of the activations in each layer so that it has a proper spread. To do that, the layer that normalizes data distribution is inserted into a neural network as the batch normalization layer



Neural network example that uses batch normalization (the batch norm layers are shown in gray)
As its name indicates, batch norm normalizes each mini-batch that is used for training. Specifically, it normalizes data so that the average is 0 and the variance is 1. The following equation shows this:

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

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

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

Here, a set of m input data, b $= \{x_1, x_2, \ldots, x_m\}$, is treated as a mini-batch and its average, μ_b , and variance, σ_b^2 , are calculated. The input data is normalized so that its average is 0 and its variance is 1 for the appropriate distribution. ε is a small value (such as 10e-7). This prevents division by 0.

Equation 6.7 simply converts the input data for a mini-batch, $\{x_1, x_2 \ldots, x_m\}$, into data with an average of 0 and a variance of 1, $\{\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_{1m}\}$. By inserting this process before (or after) the activation function

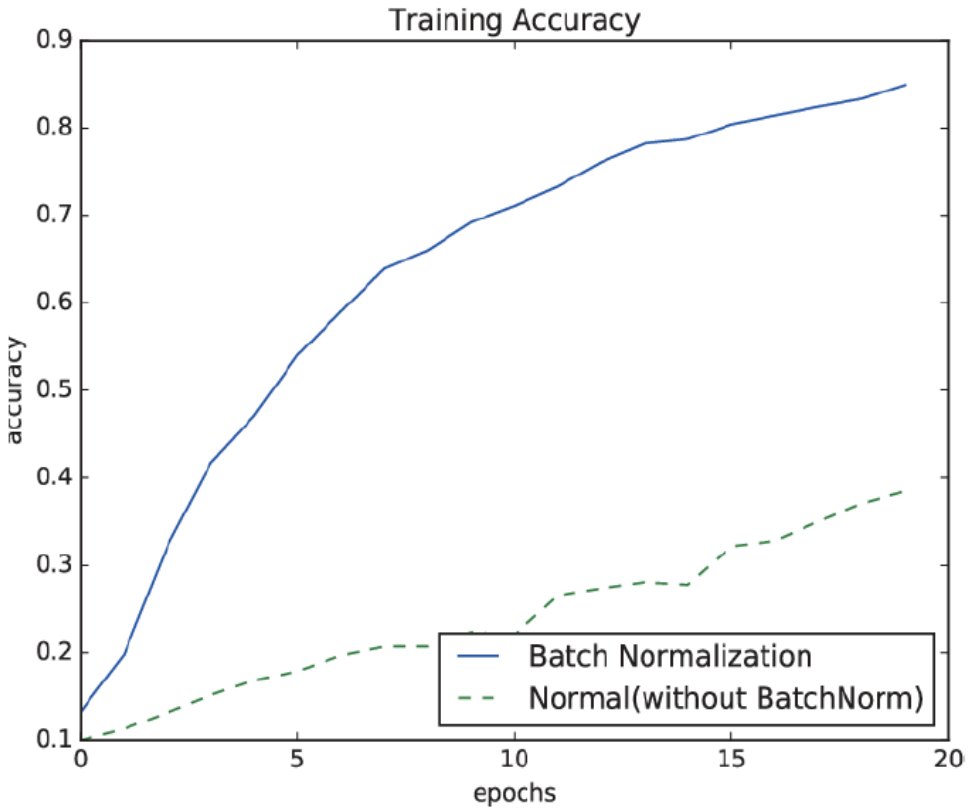
In addition, the batch norm layer converts the normalized data with a peculiar scale and shift. The following equation shows this conversion:

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

Here, γ and β are parameters. They start with γ = 1 and β = 0 and will be adjusted to the appropriate values through training.

Evaluating Batch Normalization

MNIST dataset to see how the progress of learning changes with and without the batch norm layer



Effect of batch norm – batch norm accelerates learning

2.6 Regularization

Overfitting often creates difficulties in machine learning problems. In overfitting, the model fits the training data too well and cannot properly handle other data that is not contained in the training data.

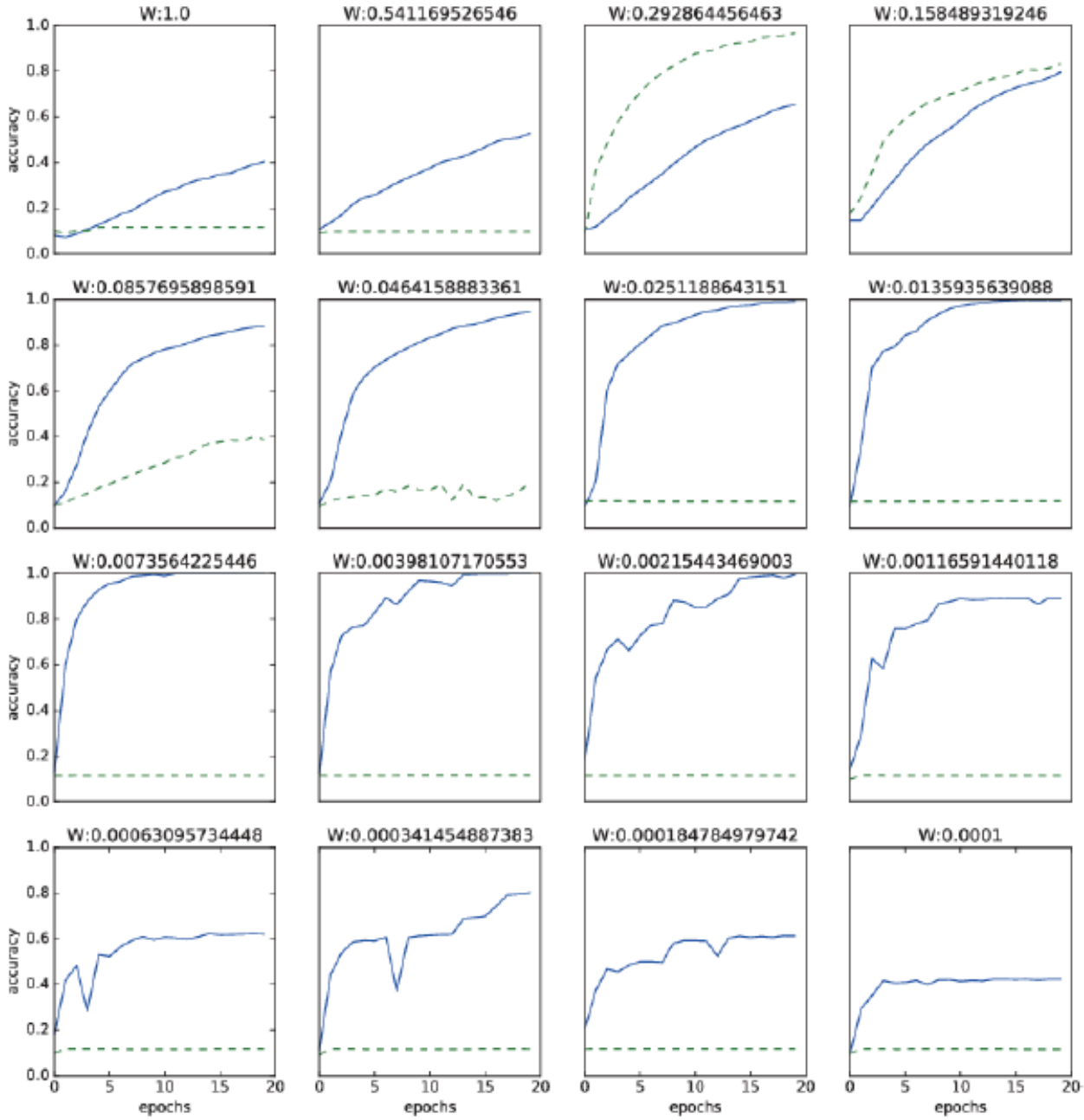
Overfitting

The main two causes of overfitting are as follows:

- The model has many parameters and is representative.
- The training data is insufficient.

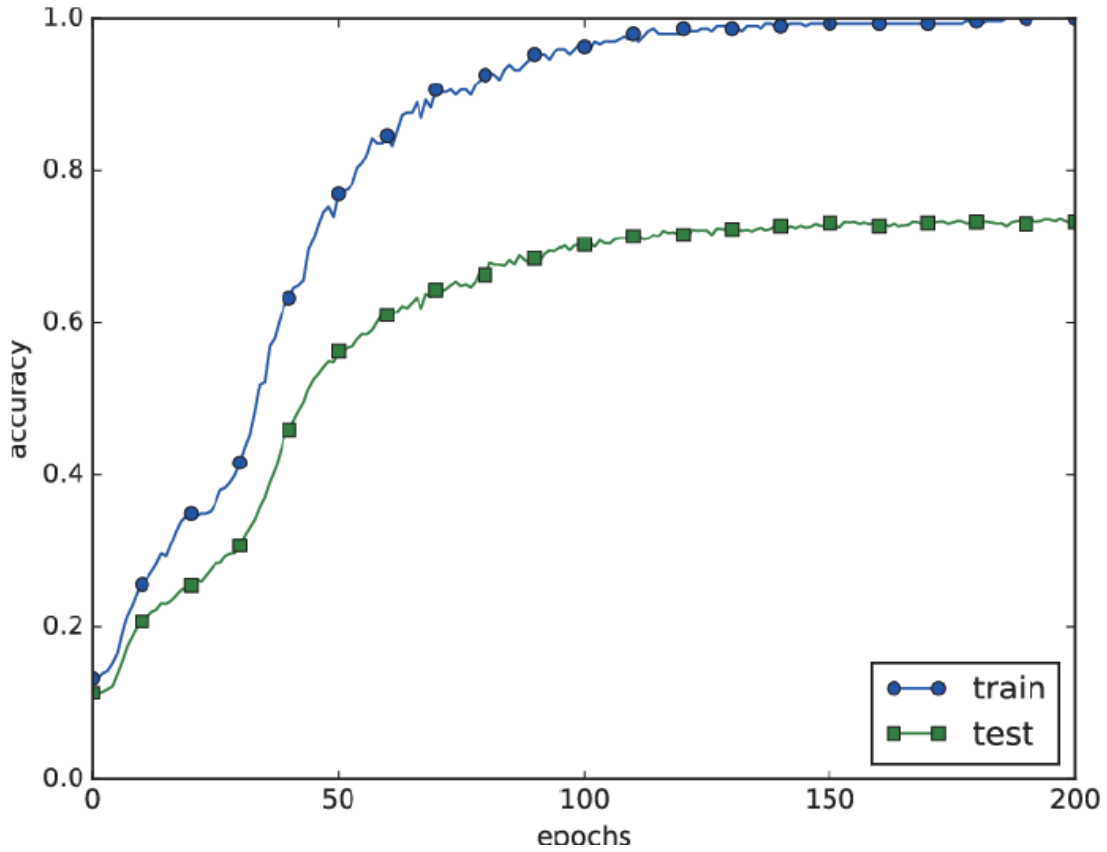
Here, we will generate overfitting by providing these two causes. Out of 60,000 pieces of training data in the MNIST dataset, only 300 are provided, and a seven-layer network is

used to increase the network's complexity. It has 100 neurons in each layer. ReLU is used as the activation function:



The solid lines show the results of using batch norm, while the dotted lines show the results without it – the title of each graph indicates the standard deviation of the initial weight values

ch06/overfit_ weight_decay.py



Transition of recognition accuracies for the training data (train) and test data (test)

The recognition accuracies that were measured using the training data reached almost 100% after 100 epochs, but the recognition accuracies on the test data are far below 100%. These large differences are caused by overfitting the training data. This graph shows that the model cannot handle general data (test data) that was not used in training properly.

Weight Decay

The **weight decay** technique has often been used to reduce overfitting. It avoids overfitting by imposing a penalty on large weights during training. Overfitting often occurs when a weight parameter takes a large value.

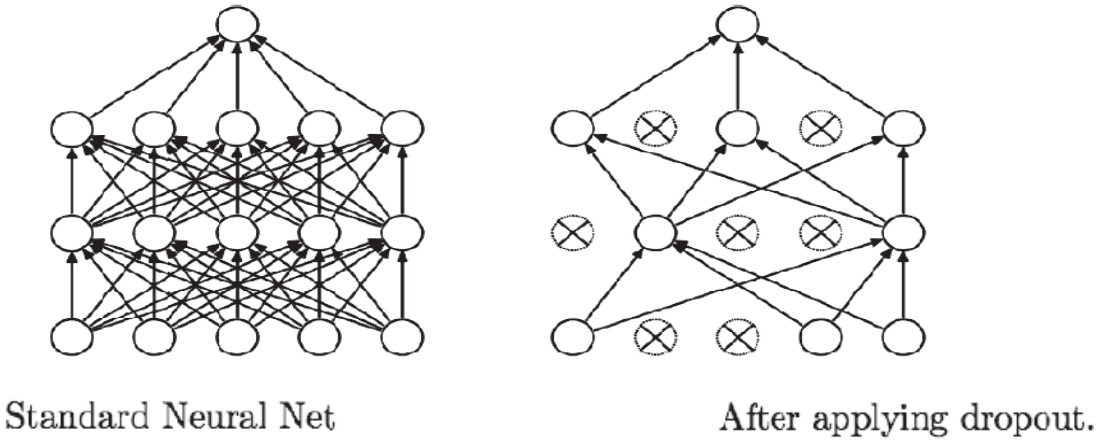
Weight decay adds $\frac{1}{2}\lambda W^2$ to the loss function for all weights. Therefore, the differential of the regularization term, λW, is added to the result of backpropagation when calculating the gradient of a weight. λ is the hyperparameter that controls the strength of regularization.

Dropout

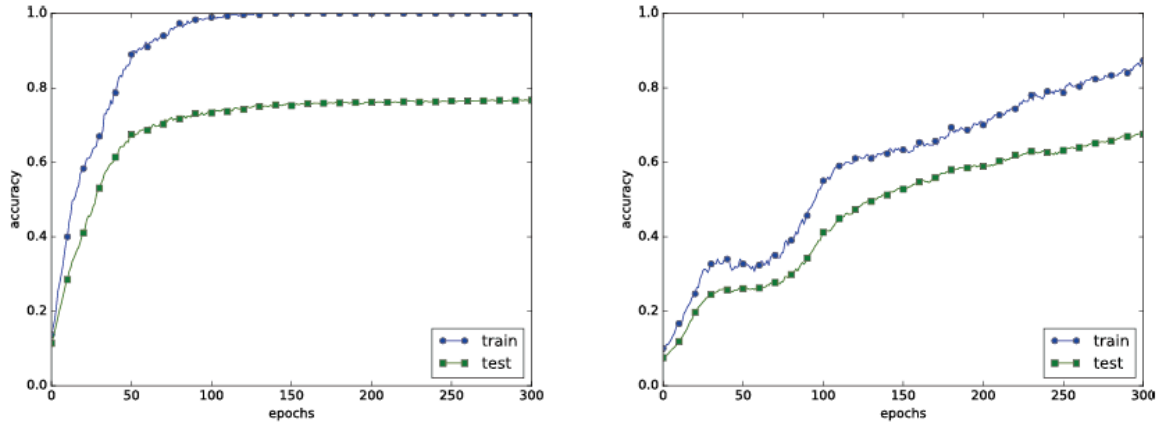
It adds the L2 norm of the weights to the loss function to reduce overfitting. Weight decay is easy to implement and can reduce overfitting to some extent. However, as a

neural network model becomes more complicated, weight decay is often insufficient. This is when the dropout technique is often used

Dropout erases neurons at random during training. During training, it selects neurons in a hidden layer at random to erase them. As shown in the following image, the erased neurons do not transmit signals. During training, the neurons to be erased are selected at random each time data flows. During testing, the signals of all the neurons are propagated. The output of each neuron is multiplied by the rate of the erased neurons during training:



using dropout reduces the difference between the recognition accuracies of training data and test data. It also indicates that the recognition accuracy of the training data has not reached 100%. Due to this, you can use dropout to reduce overfitting, even in a representative network:



The left-hand image shows the experiment without dropout, while the right-hand image shows the experiment with dropout (dropout_rate=0.15)

2.7 Validating Hyper parameters

A neural network uses many hyper parameters, as well as parameters such as weights and biases. The hyper parameters here include the number of neurons in each layer,

batch size, the learning rate for updating parameters, and weight decay. Setting the Hyper parameters to inappropriate values deteriorates the performance of the model. The values of these Hyper parameters are very important, but determining them usually requires a lot of trial and error.

Validation Data

The training data is used to train a network, while the test data is used to evaluate generalization performance. Thus, you can determine whether or not the network conforms too well only to the training data (that is, whether overfitting occurs) and how large the generalization performance is.

You must not use test data to evaluate the performance of hyper parameters.

Why can't we use test data to evaluate the performance of hyper parameters? Well, if we use test data to adjust hyper parameters, the hyper parameter values will overfit the test data. In other words, it uses test data to check that the hyper parameter values are "good," so the hyper parameter values are adjusted so that they only fit the test data. Here, the model may provide low generalization performance and cannot fit other data.

Therefore, we need to use verification data (called **validation data**) to adjust them. This validation data is used to evaluate the quality of our hyper parameters.

Training data is used for learning parameters (weights and biases). Validation data is used to evaluate the performance of hyper parameters. Test data is used (once, ideally) at the end of training to check generalization performance.

Some datasets provide training data, validation data, and test data separately. Some provide only training data and test data, while some provide only one type of data. In that case, you must separate the data manually.

Optimizing Hyperparameters

What is important when optimizing hyperparameters is to gradually narrow down the range where "good" hyperparameters values exist. To do this, we will set a broad range initially, select hyperparameters at random from the range (sampling), and use the sampled values to evaluate the recognition accuracy. Next, we will repeat these steps several times and observe the result of the recognition accuracy. Based on the result, we will narrow down the range of "good" hyperparameters values. By repeating this procedure, we can gradually limit the range of appropriate hyperparameters.

when optimizing hyperparameters, deep learning takes a lot of time (even a few days or weeks). Therefore, any hyperparameters that seem inappropriate must be abandoned

while searching for them. When optimizing hyperparameters, it is effective to reduce the size of epoch for training to shorten the time that one evaluation takes.

Step 0

Specify the range of the hyperparameters.

Step 1

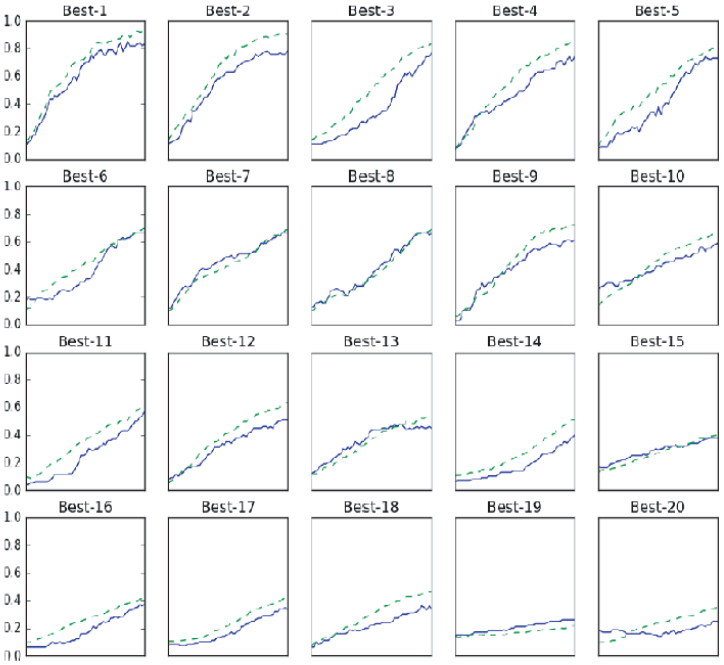
Sample the hyperparameters from the range at random.

Step 2

Use the hyperparameter values sampled in *Step 1* for training and use the validation data to evaluate the recognition accuracy (set small epochs).

Step 3

Repeat *steps 1* and *2* a certain number of times (such as 100 times) and narrow down the range of hyperparameters based on the result of the recognition accuracy. When the range is narrowed down to some extent, select one hyperparameter value from it. This is one practical approach to optimizing hyperparameters.



The solid lines show the recognition accuracies of the validation data, while the dotted lines show the recognition accuracies of the training data