学习深度学习之前，你需要知道的

---

*本文章旨在为准备学习深度学习且具备基础编程能力的人提供最简单和基础的理论模型。注意：文章中的所有内容来自于《深度学习入门：基于**Python**的理论与实现》（斋藤康毅著）*

本书将使用下列编程语言和库：

- *Python 3.x*
- *NumPy*
- *Matplotlib*

Python 入门

Python是什么

Python 是最适合数据科学领域的编程语言，而且是开源的，可以免费地自由使用。Python 除了高性能之外，凭借着 NumPy、SciPy 等优秀的数值计算、统计分析库，在数据科学领域占有不可动摇的地位。

Python的安装

Python 的安装方法有很多种，推荐使用 Anaconda 这个发行版。发行版集成了必要的库，使用户可以一次性完成安装。Anaconda 是一个侧重于数据分析的发行版，前面说的 NumPy、Matplotlib 等有助于数据分析的库都包含在其中 。

python基础知识

问AI自学，要求掌握：算术计算、数据类型、变量、列表、字典、布尔型、if语句、for语句、函数、类。

Numpy

深度学习中需要数组和矩阵的计算，Numpy的数组类提供了很多便捷的方法。如果使用Anaconda，则已安装此库。

Numpy相关知识问AI自学，要求掌握：数组、算术运算、广播、访问元素。

Matplotlib

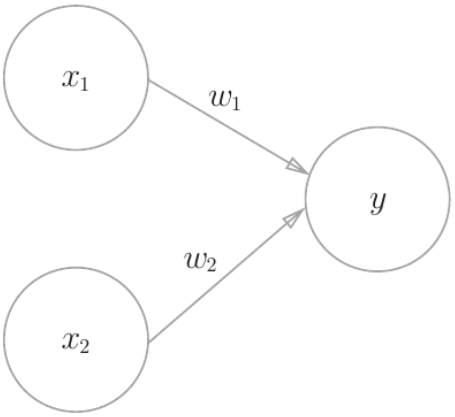
可以使用 matplotlib 的 pyplot 模块绘制图形，帮助我们直观地观察数据反映的趋势。

Matplotlib相关知识问AI自学，要求掌握：基础绘图、显示图像。

感知机

感知机是由美国学者 Frank Rosenblatt 在 1957 年提出来的。为感知机也是作为神经网络（深度学习）的起源的算法。

感知机是什么



这是一个接收两个输入信号的感知机的例子。x1、x2 是输入信号，y 是输出信号，w1、w2 是权重（w 是 weight 的首字母）。图中的○称为“神经元”或者“节点”。输入信号被送往神经元时，会被分别乘以固定的权重（w1x1、w2x2 ）。神经元会计算传送过来的信号的总和，只有当这个总和超过了某个界限值时，才会输出 1，否则输出0。

把上述内容用数学式来表示：

$$y = \begin{cases} 0 & (w_1x_1 + w_2x_2 \leq \theta) \\ 1 & (w_1x_1 + w_2x_2 > \theta) \end{cases}$$

简单逻辑电路与感知机

感知机可以实现与门、或门和非门。下面是一个实现与门的简单例子：

```
def AND(x1, x2):
    w1, w2, theta = 0.5, 0.5, 0.7
    tmp = x1*w1 + x2*w2
    if tmp <= theta:
        return 0
    elif tmp > theta:
        return 1
```

导入权重和偏执

我们把之前的表达式修改一下形式：

$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leq 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases}$$

感知机会计算输入信号和权重的乘积，然后加上偏置，如果这个值大于 0 则输出 1，否则输出 0。b 是被称为偏置的参数，用于控制神经元被激活的容易程度；而 w1 和 w2是表示各个信号的权重的参数，用于控制各个信号的重要性。

现在，我们用使用权重和偏置，可以像下面这样实现与门：

```
def AND(x1, x2):
    x = np.array([x1, x2])
    w = np.array([0.5, 0.5])
    b = -0.7
    tmp = np.sum(w*x) + b
    if tmp <= 0:
        return 0
    else:
        return 1
```

同理，与非门和或门：

```
def NAND(x1, x2):
    x = np.array([x1, x2])
    w = np.array([-0.5, -0.5]) # 仅权重和偏置与 AND 不同!
    b = 0.7
    tmp = np.sum(w*x) + b
```

```
    if tmp <= 0:
        return 0
    else:
        return 1

def OR(x1, x2):
    x = np.array([x1, x2])
    w = np.array([0.5, 0.5]) # 仅权重和偏置与 AND 不同!
    b = -0.2
    tmp = np.sum(w*x) + b
    if tmp <= 0:
        return 0
    else:
        return 1
```

像这样，偏置的值决定了神经元被激活的容易程度。另外，这里我们将 w1 和 w2 称为权重，将 b 称为偏置，但是根据上下文，有时也会将 b、w1、w2 这些参数统称为权重。

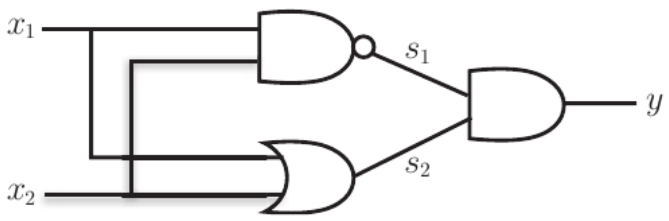
感知机的局限性

前面介绍的感知机是无法实现异或门。感知机的局限性就在于它只能表示由一条直线分割的空间。

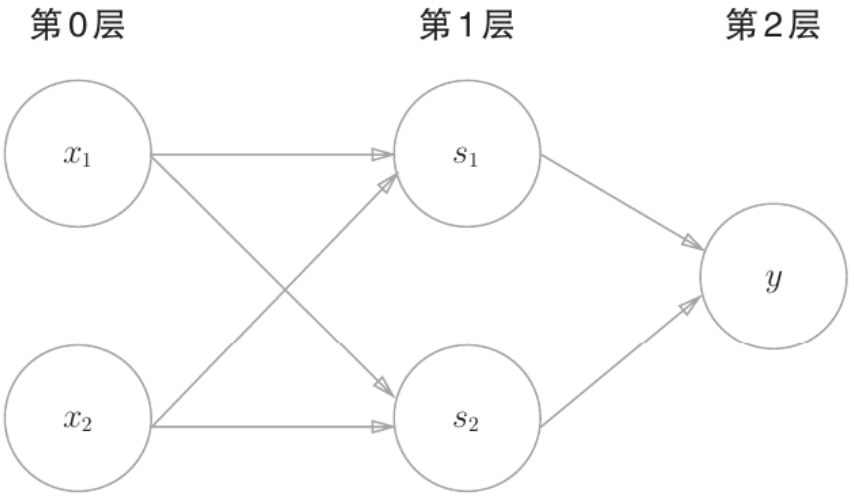
多层感知机

感知机可以通过“叠加层”的方式实现一些复杂的逻辑。下面是实现异或门的例子：

```
def XOR(x1, x2):
    s1 = NAND(x1, x2)
    s2 = OR(x1, x2)
    y = AND(s1, s2)
    return y
```



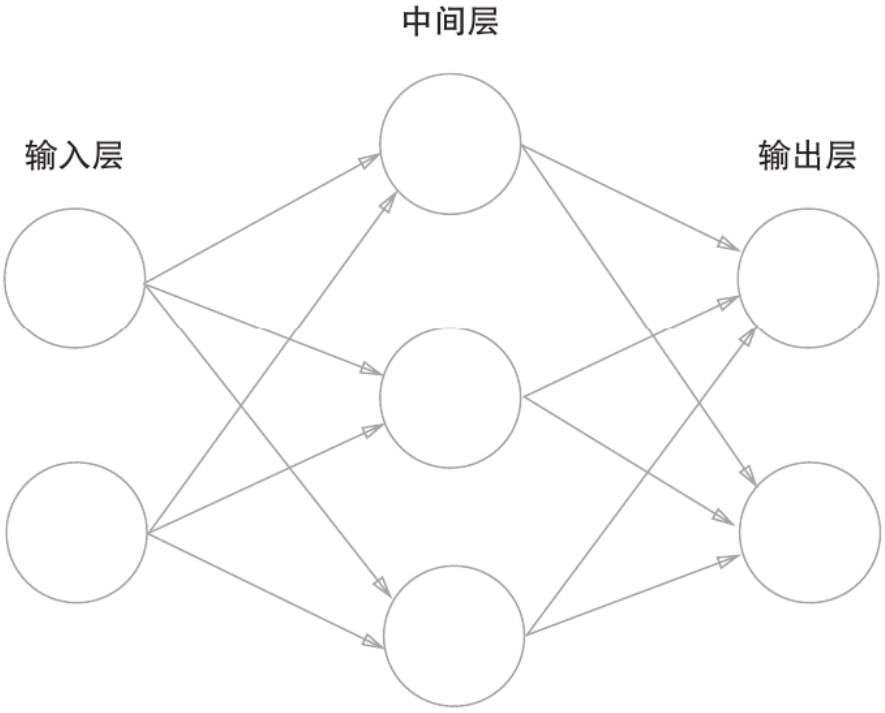
与门、或门是单层感知机，而异或门是 2 层感知机。叠加了多层的感知机也称为多层感知机（multi-layered perceptron）。这里，将最左边的一列称为第 0 层，中间的一列称为第 1 层，最右边的一列称为第 2 层。



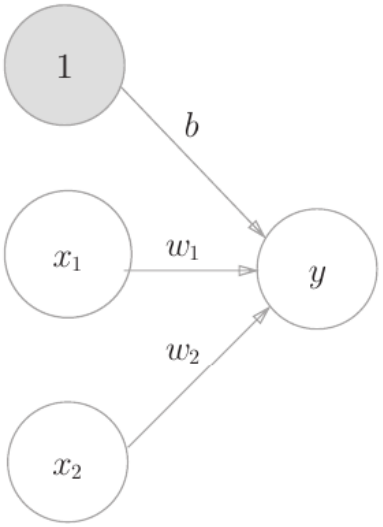
感知机通过叠加层能够进行非线性的表示，理论上还可以表示计算机进行的处理。实际上只需要通过与非门的组合，就能再现计算机进行的处理。

神经网络

神经网络的一个重要性质是它可以自动地从数据中学习到合适的权重参数，而不是像我们之前需要人为设定权重的值，这在系统，特别是复杂的系统中是极为必要的，后者可能存在上亿个参数。



上图是一个神经网络的例子，神经网络的形状类似上一章的感知机。实际上，就神经元的连接方式而言，与上一章的感知机并没有任何差异。我们把最左边的一列称为输入层，最右边的一列称为输出层，中间的一列称为中间层。中间层有时也称为隐藏层。“隐藏”一词的意思是，隐藏层的神经元（和输入层、输出层不同）肉眼看不见。



激活函数

如果我们把上面的表达式改写为一个函数：

$$y = h(b + w_1x_1 + w_2x_2)$$

$$h(x) = \begin{cases} 0 & (x \leq 0) \\ 1 & (x > 0) \end{cases}$$

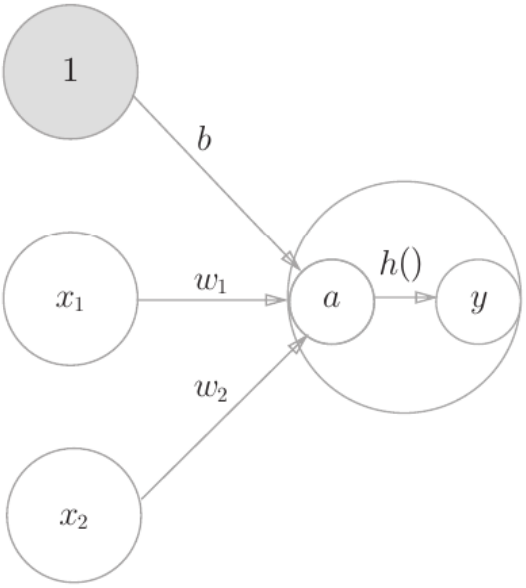
h（x）函数会将输入信号的总和转换为输出信号，这种函数一般称为激活函数（activation function）。如“激活”一词所示，激活函数的作用在于决定如何来激活输入信号的总和。

若进一步改写：

$$a = b + w_1x_1 + w_2x_2$$

$$y = h(a)$$

则之前的神经元可以这样表示：



激活函数是连接感知机和神经网络的桥梁。

实际上，我们之前的表达式可以说感知机中使用了阶跃函数作为激活函数。激活函数以阈值为界，一旦输入超过阈值，就切换输出。如果将激活函数从阶跃函数换成其他函数，就可以进入神经网络的世界了。

sigmoid 函数

神经网络中经常使用的一个激活函数就是 sigmoid 函数（sigmoid function）。

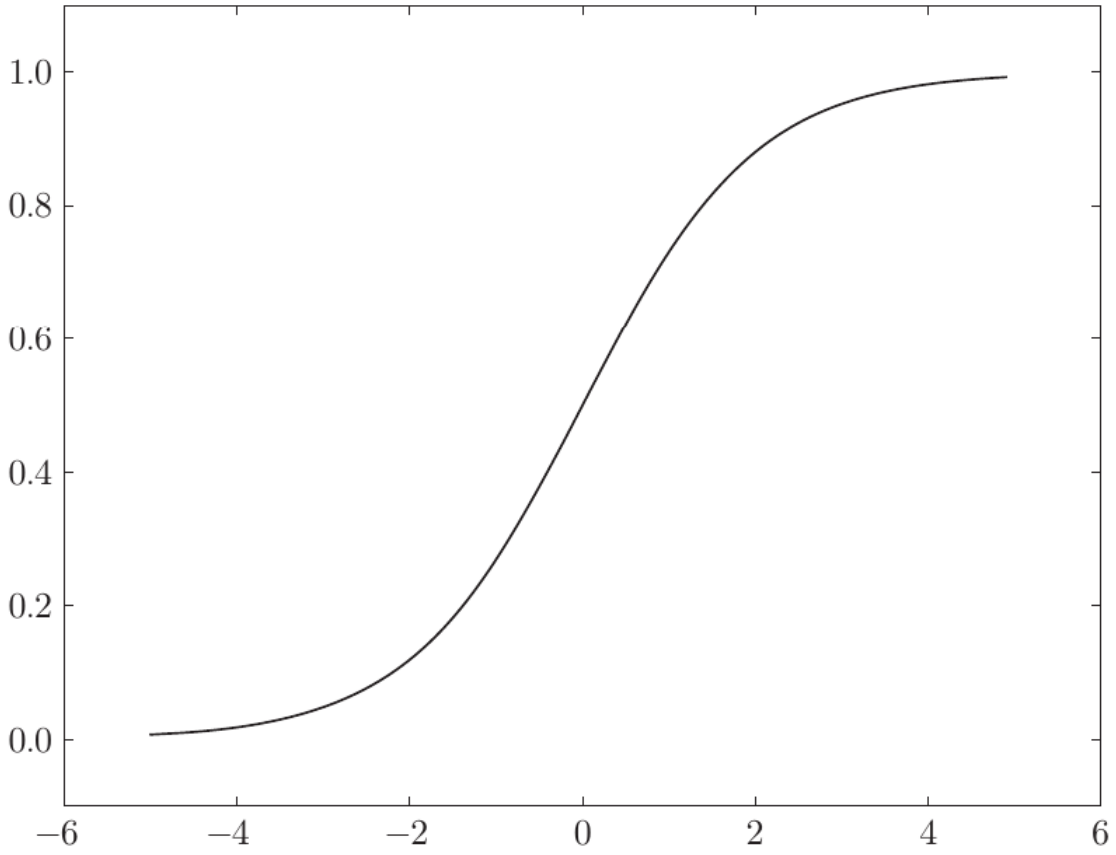
$$h(x) = \frac{1}{1 + exp(-x)}$$

下面，我们来实现 sigmoid 函数。用 Python 可以像下面这样表示的 sigmoid 函数。

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

我们把sigmoid 函数画在图上。

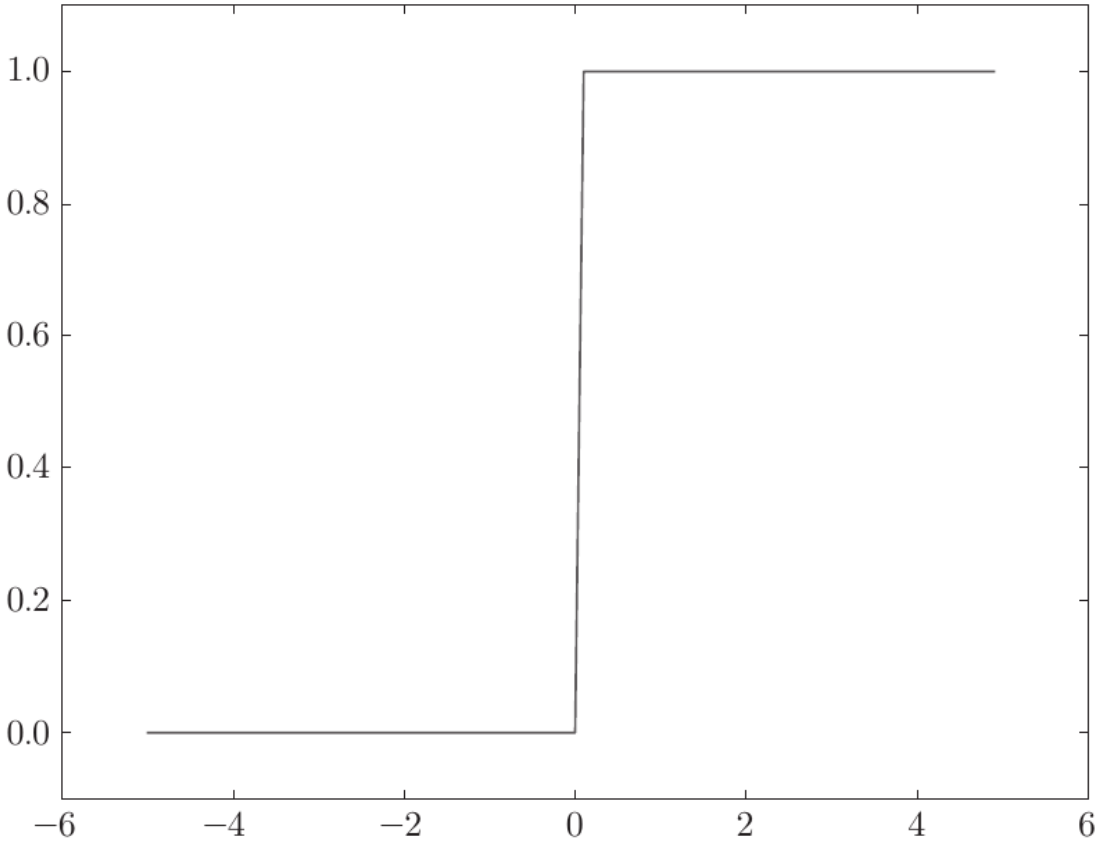
```
x = np.arange(-5.0, 5.0, 0.1)
y = sigmoid(x)
plt.plot(x, y)
plt.ylim(-0.1, 1.1) # 指定 y 轴的范围
plt.show()
```



阶跃函数

这里我们试着用 Python 画出阶跃函数的图。

```
def step_function(x):
    y = x > 0
    return y.astype(np.int)

import numpy as np
import matplotlib.pylab as plt
def step_function(x):
    return np.array(x > 0, dtype=np.int)
x = np.arange(-5.0, 5.0, 0.1)
y = step_function(x)
plt.plot(x, y)
plt.ylim(-0.1, 1.1) # 指定 y 轴的范围
plt.show()
```



sigmoid 函数和阶跃函数的比较

sigmoid 函数是一条平滑的曲线，输出随着输入发生连续性的变化。而阶跃函数以 0 为界，输出发生急剧性的变化。sigmoid 函数的平滑性对神经网络的学习具有重要意义。

另一个不同点是，相对于阶跃函数只能返回 0 或 1，sigmoid 函数可以返回 0.731 . . .、0.880 . . . 等实数（这一点和刚才的平滑性有关）。也就是说，感知机中神经元之间流动的是 0 或 1 的二元信号，而神经网络中流动的是连续的实数值信号。

第一个相同点是，当输入信号为重要信息时，阶跃函数和 sigmoid 函数都会输出较大的值；当输入信号为不重要的信息时，两者都输出较小的值。还有一个共同点是，不管输入信号有多小，或者有多大，输出信号的值都在 0 到 1 之间。

非线性函数

神经网络的激活函数必须使用非线性函数。换句话说，激活函数不能使用线性函数。为什么不能使用线性函数呢？因为使用线性函数的话，加深神经网络的层数就没有意义了。

ReLU函数

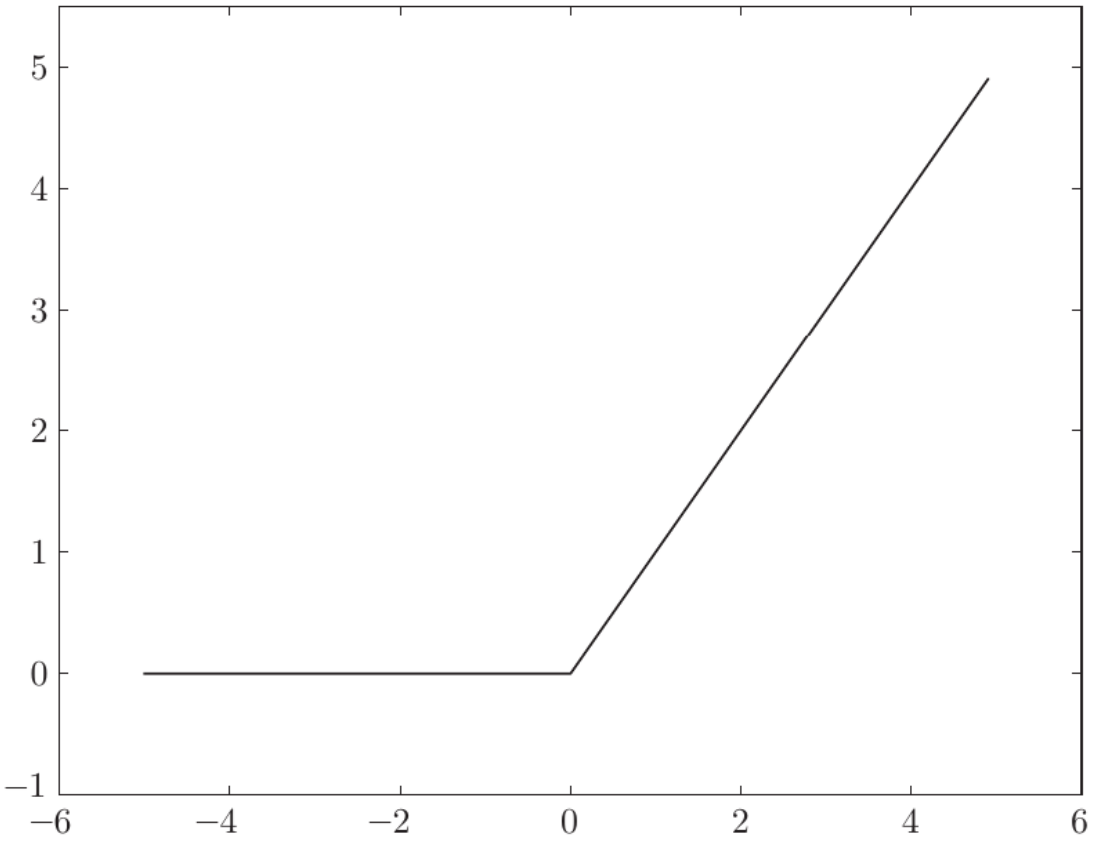
在神经网络发展的历史上，sigmoid 函数很早就开始被使用了，而最近则主要使用 ReLU（Rectified Linear Unit）函数

ReLU 函数在输入大于 0 时，直接输出该值；在输入小于等于 0 时，输出 0。

$$h(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$

实现这个函数很简单：

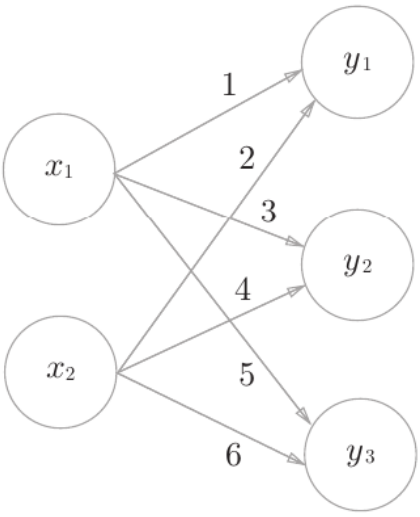
```
def relu(x):
    return np.maximum(0, x)
```



多维数组的计算

自行AI学习，要求掌握：数组的维数和形状、矩阵乘法。

三层神经网络的实现



$$\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

$$\boldsymbol{X} \quad \boldsymbol{W} = \boldsymbol{Y}$$

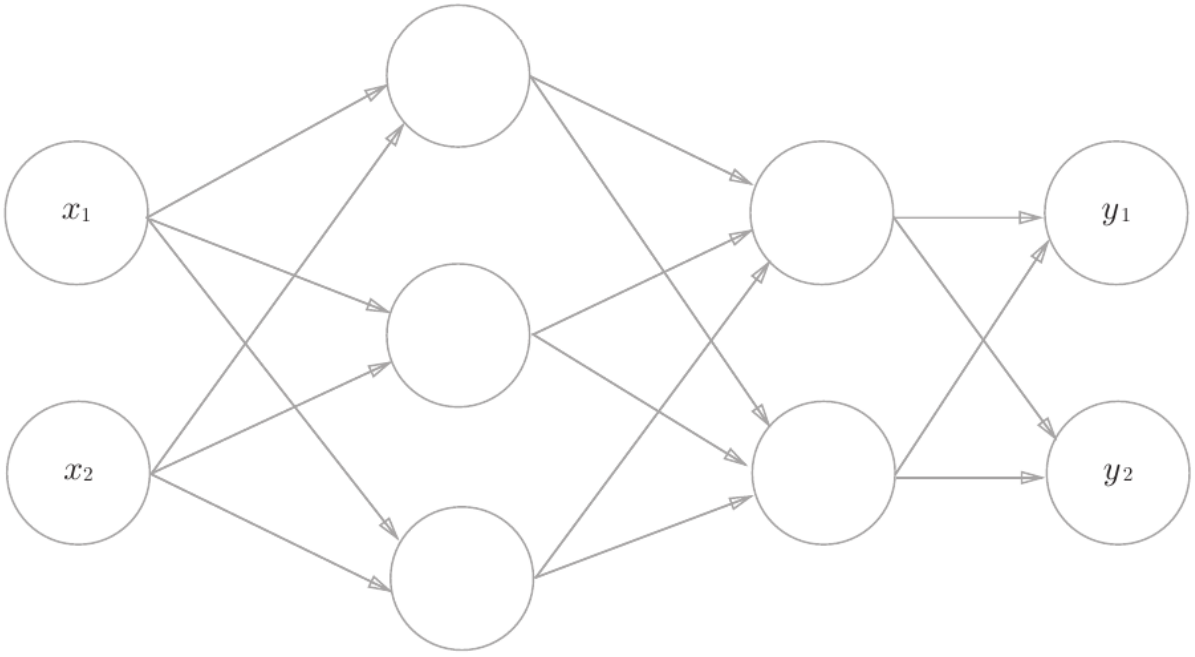
$$2 \quad 2 \times 3 \quad 3$$

一致

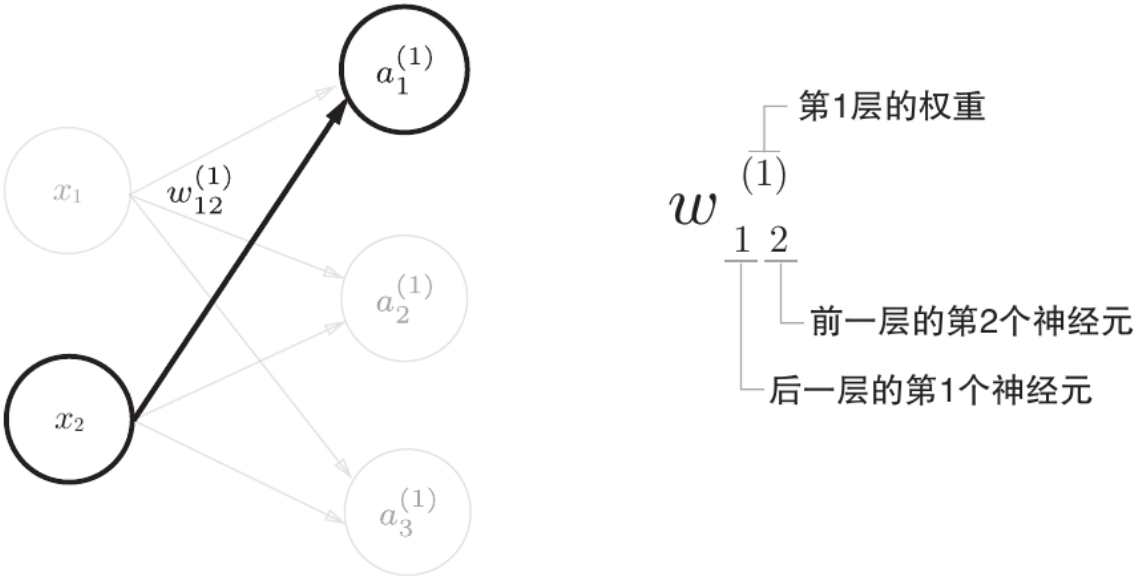
实现该神经网络时，要注意 X、W、Y 的形状，特别是 X 和 W 的对应维度的元素个数是否一致，这一点很重要。

```
>>> X = np.array([1, 2])
>>> X.shape
(2,)
>>> W = np.array([[1, 3, 5], [2, 4, 6]])
>>> print(W)
[[1 3 5]
[2 4 6]]
>>> W.shape
(2, 3)
>>> Y = np.dot(X, W)
>>> print(Y)
[ 5 11 17]
```

在代码实现方面，使用上一节介绍的 NumPy 多维数组，可以用很少的代码完成神经网络的前向处理。



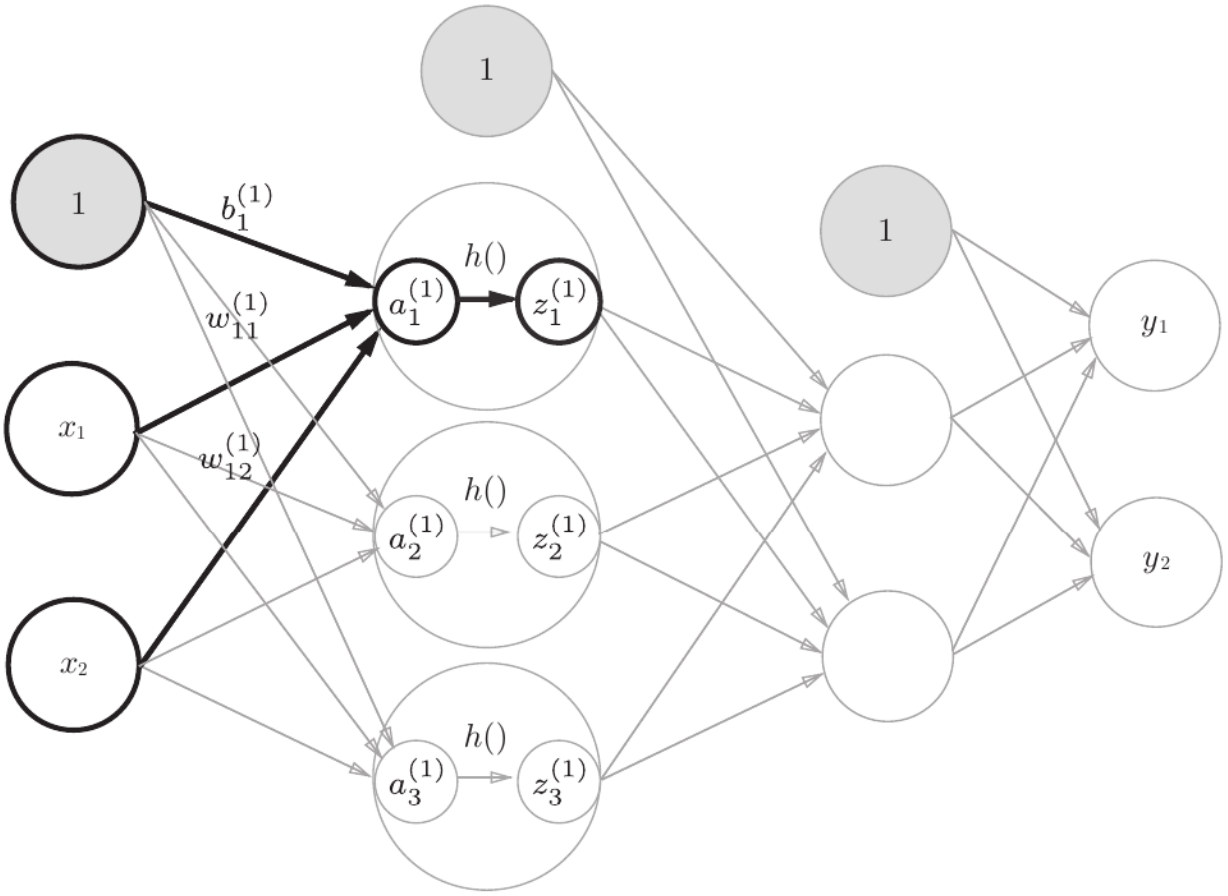
首先，我们引入一些便于理解的符号D



具体的数学推到验证较为简单，可自行验证，这里给出完整代码：

```
def init_network():
    network = {}
    network['W1'] = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
    network['b1'] = np.array([0.1, 0.2, 0.3])
    network['W2'] = np.array([[0.1, 0.4], [0.2, 0.5], [0.3,
0.6]])
    network['b2'] = np.array([0.1, 0.2])
    network['W3'] = np.array([[0.1, 0.3], [0.2, 0.4]])
    network['b3'] = np.array([0.1, 0.2])
    return network
```

```python
def forward(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']
    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = identity_function(a3)
    return y
network = init_network()
x = np.array([1.0, 0.5])
y = forward(network, x)
print(y) # [ 0.31682708 0.69627909]
```



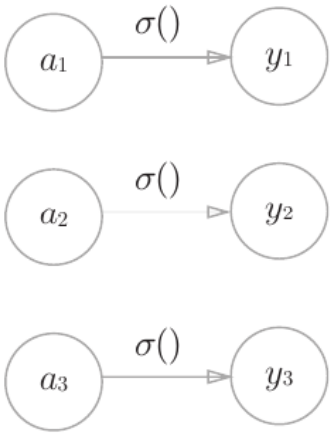
输出层设计

神经网络可以用在分类问题和回归问题上，不过需要根据情况改变输出层的激活函数。一般而言，回归问题用恒等函数，分类问题用 softmax 函数。

机器学习的问题大致可以分为分类问题和回归问题。分类问题是数据属于哪一个类别的问题。比如，区分图像中的人是男性还是女性的问题就是分类问题。而回归问题是根据某个输入预测一个（连续的）数值的问题。比如，根据一个人的图像预测这个人的体重的问题就是回归问题（类似“57.4kg”这样的预测）。

恒等函数和 softmax 函数

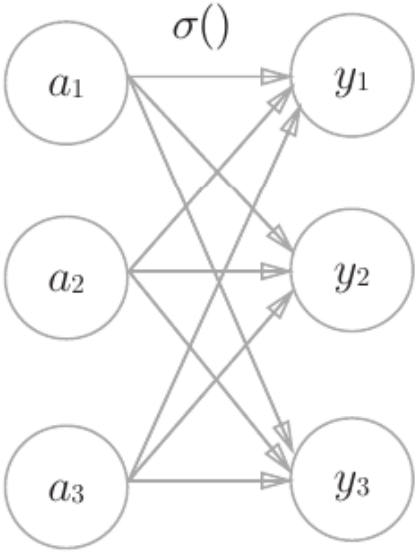
恒等函数会将输入按原样输出，对于输入的信息，不加以任何改动地直接输出。



分类问题中使用的 softmax 函数可以用下面的式表示。:

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)}$$

用图表示 softmax 函数的话,输出通过箭头与所有的输入信号相连。这是因为，输出层的各个神经元都受到所有输入信号的影响。



实现 softmax 函数时的注意事项

softmax 函数的实现中要进行指数函数的运算，但是此时指数函数的值很容易变得非常大。e^1000 的结果会返回
一个表示无穷大的 inf 。如果在这些超大值之间进行除法运算，结果会出现“不确定”的情况，即溢出问题。

在进行 softmax 的指数函数的运算时，加上（或者减去）某个常数并不会改变运算的结果，为了防止溢出，一般会使用输入信号中的最大值。（数学推导自行完成）下面是一个例子：

```
>>> a = np.array([1010, 1000, 990])
>>> np.exp(a) / np.sum(np.exp(a)) # softmax 函数的运算
array([ nan, nan, nan])
 # 没有被正确计算
>>>
>>> c = np.max(a) # 1010
>>> a - c
array([ 0, -10, -20])
>>>
>>> np.exp(a - c) / np.sum(np.exp(a - c))
array([ 9.99954600e-01,
 4.53978686e-05,
 2.06106005e-09])
```

softmax 函数的输出是 0.0 到 1.0 之间的实数。并且，softmax函数的输出值的总和是 1。输出总和为 1 是 softmax 函数的一个重要性质。正因为有了这个性质，我们才可以把 softmax 函数的输出解释为“概率”。

一般而言，神经网络只把输出值最大的神经元所对应的类别作为识别结果。并且，即便使用 softmax 函数，输出值最大的神经元的位置也不会变。因此，神经网络在进行分类时，输出层的 softmax 函数可以省略。在实际的问题中，由于指数函数的运算需要一定的计算机运算量，因此输出层的 softmax 函数一般会被省略。

求解机器学习问题的步骤可以分为“学习”A 和“推理”两个阶段。首先，在学习阶段进行模型的学习B，然后，在推理阶段，用学到的模型对未知的数据进行推理（分类）。如前所述，推理阶段一般会省略输出层的softmax 函数。在输出层使用softmax 函数是因为它和神经网络的学习有关系。

手写数字识别（神经网络的前向传播）

本小结不涉及相关代码，只讲述重要概念

MNIST数据集

MNIST 是机器学习领域最有名的数据集之一，被应用于从简单的实验到发表的论文研究等各种场合。MNIST 的图像数据是 28 像素 × 28 像素的灰度图像（1 通道），各个像素的取值在 0 到 255 之间。每个图像数据都相应地标有“7”“2”“1”等标签。训练图像有 6 万张，测试图像有 1 万张，这些图像可以用于学习和推理。

one-hot 表示是仅正确解标签为1，其余皆为0 的数组，就像 [0,0,1,0,0,0,0,0,0,0] 这样

Python 有 pickle 这个便利的功能。这个功能可以将程序运行中的对象保存为文件。如果加载保存过的 pickle 文件，可以立刻复原之前程序运行中的对象。用于读入 MNIST 数据集的 load_mnist() 函数内部也使用了 pickle 功能（在第 2 次及以后读入时）。利用 pickle 功能，可以高效地完成 MNIST 数据的准备工作。

把数据限定到某个范围内的处理称为正规化 （normalization）。此外，对神经网络的输入数据进行某种既定的转换称为预处理 （pre-processing）。这里，作为对输入图像的一种预处理，我们进行了正规化。

批处理

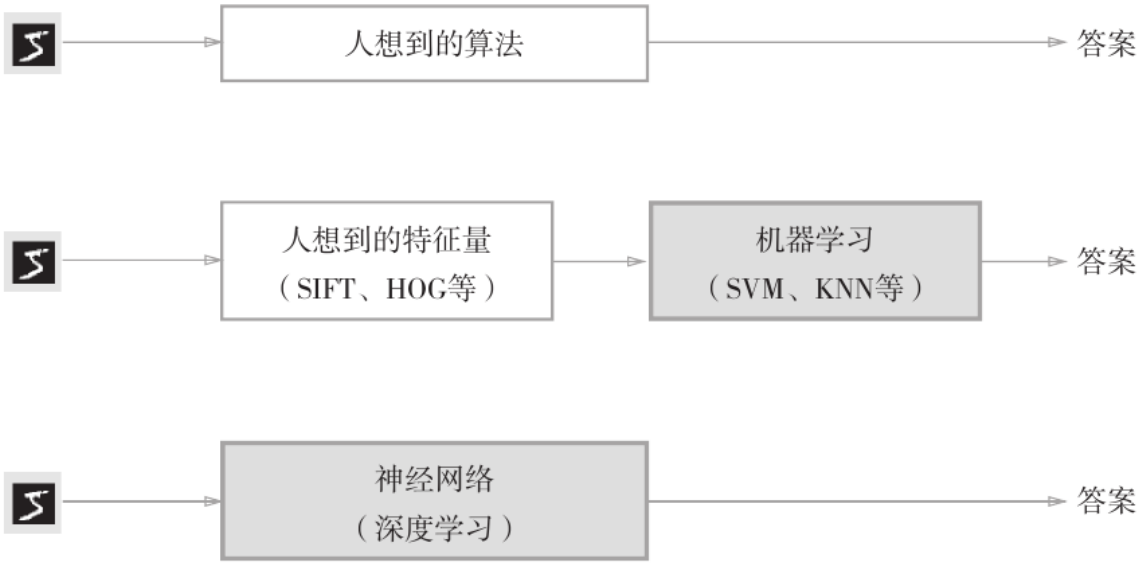
现在我们来考虑打包输入多张图像的情形。比如，我们想一次性打包处理 100 张图像。

批处理对计算机的运算大有利处，可以大幅缩短每张图像的处理时间。那么为什么批处理可以缩短处理时间呢？这是因为大多数处理数值计算的库都进行了能够高效处理大型数组运算的最优化。并且，在神经网络的运算中，当数据传送成为瓶颈时，批处理可以减轻数据总线的负荷（严格地讲，相对于数据读入，可以将更多的时间用在计算上）。也就是说，批处理一次性计算大型数组要比分开逐步计算各个小型数组速度更快。

神经网络的学习

所谓“从数据中学习”，是指可以由数据自动决定权重参数的值。在神经网络学习中，不需要人为设计合适的特征量。

神经网络的优点是对所有的问题都可以用同样的流程来解决。比如，不管要求解的问题是识别 5，还是识别狗，抑或是识别人脸，神经网络都是通过不断地学习所提供的数据，尝试发现待求解的问题的模式。也就是说，与待处理的问题无关，神经网络可以将数据直接作为原始数据，进行“端对端”的学习。



训练数据和测试数据

机器学习中，一般将数据分为训练数据和测试数据两部分来进行学习和实验等。首先，使用训练数据进行学习，寻找最优的参数；然后，使用测试数据评价训练得到的模型的实际能力。

为了正确评价模型的泛化能力，就必须划分训练数据和测试数据。另外，训练数据也可以称为监督数据。泛化能力是指处理未被观察过的数据（不包含在训练数据中的数据）的能力。获得泛化能力是机器学习的最终目标。

只对某个数据集过度拟合的状态称为过拟合 （over fitting）。避免过拟合也是机器学习的一个重要课题。

损失函数

神经网络的学习通过某个指标表示现在的状态。然后，以这个指标为基准，寻找最优权重参数。神经网络的学习中
所用的指标称为损失函数 （loss function） 。这个损失函数可以使用任意函数，但一般用均方误差和交叉熵误差等。

均方误差

$$E = \frac{1}{2}\sum_k (y_k - t_k)^2$$

这里，yk 是表示神经网络的输出，tk 表示监督数据，k 表示数据的维数。

交叉熵误差

$$E = -\sum_k t_k \log y_k$$

yk 是神经网络的输出，tk 是正确解标签。并且， tk 中只有正确解标签的索引为 1，其他均为 0 （one-hot 表示）。

交叉熵误差的值是由正确解标签所对应的输出结果决定的。

在函数内部在计算 np.log 时，加上了一个微小值 delta。这是因为，当出现 np.log(0) 时，np.log(0) 会变为负无限大的 -inf，这样一来就会导致后续计算无法进行。作为保护性对策，添加一个微小值可以防止负无限大的发生。

mini-batch学习

前面介绍的损失函数的例子中考虑的都是针对单个数据的损失函数。如果要求所有训练数据的损失函数的总和，以交叉熵误差为例，可以写成下面的式：

$$E = -\frac{1}{N}\sum_n \sum_k t_{nk} \log y_{nk}$$

这里 , 假设数据有 N 个，tnk 表示第 n 个数据的第 k 个元素的值（ynk 是神经网络的输出，tnk 是监督数据）。

从 60000 个训练数据中随机选择 100 笔，再用这 100 笔数据进行学习。这种学习方式称为 mini-batch 学习。

为什么要设定损失函数

在进行神经网络的学习时，不能将识别精度作为指标。因为如果以识别精度为指标，则参数的导数在绝大多数地方都会变为 0。

阶跃函数的导数在绝大多数地方（除了 0 以外的地方）均为 0。也就是说，如果使用了阶跃函数，那么即便将损失函数作为指标，参数的微小变化也会被阶跃函数抹杀，导致损失函数的值不会产生任何变化。而sigmoid 函数的导数在任何地方都不为 0。这对神经网络的学习非常重要。得益于这个斜率不会为 0 的性质，神经网络的学习得以正确进行。

梯度法

在梯度法中，函数的取值从当前位置沿着梯度方向前进一定距离，然后在新的地方重新求梯度，再沿着新梯度方向前进，
如此反复，不断地沿梯度方向前进。像这样，通过不断地沿梯度方向前进，逐渐减小函数值的过程就是梯度法（gradient method）。梯度法是解决机器学习中最优化问题的常用方法，特别是在神经网络的学习中经常被使用。

这里需要注意的是，梯度表示的是各点处的函数值减小最多的方向。因此，无法保证梯度所指的方向就是函数的最小值或者真正应该前进的方向。实际上，在复杂的函数中，梯度指示的方向基本上都不是函数值最小处。

我们尝试用数学式来表示梯度法：

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

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

η 表示更新量，在神经网络的学习中，称为学习率（learningrate）。学习率决定在一次学习中，应该学习多少，以及在多大程度上更新参数。

实验结果表明，学习率过大的话，会发散成一个很大的值；反过来，学习率过小的话，基本上没怎么更新就结束了。也就是说，设定合适的学习率是一个很重要的问题。

像学习率这样的参数称为超参数。这是一种和神经网络的参数（权重和偏置）性质不同的参数。相对于神经网络的权重参数是通过训练数据和学习算法自动获得的，学习率这样的超参数则是人工设定的。一般来说，超参数需要尝试多个值，以便找到一种可以使学习顺利进行的设定。

神经网络的梯度

神经网络的学习也要求梯度。这里所说的梯度是指损失函数关于权重参数的梯度。比如，有一个只有一个形状为 2 × 3 的权重 W 的神经网络，损失函数用 L 表示。用数学式表示的话，如下所示：

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

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

求出神经网络的梯度后，接下来只需根据梯度法，更新权重参数即可。

学习算法的实现

1. 前提：神经网络存在合适的权重和偏置，调整权重和偏置以便拟合训练数据的过程称为“学习”。神经网络的学习分成下面 4 个步骤。
2. 步骤一（mini-batch）：从训练数据中随机选出一部分数据，这部分数据称为 mini-batch。我们的目标是减小 mini-batch 的损失函数的值。
3. 步骤二（计算梯度）：为了减小 mini-batch 的损失函数的值，需要求出各个权重参数的梯度。梯度表示损失函数的值减小最多的方向。
4. 步骤三（更新参数）：将权重参数沿梯度方向进行微小更新。
5. 步骤四（重复）：重复步骤 1、步骤 2、步骤 3。

神经网络的学习按照上面 4 个步骤进行。这个方法通过梯度下降法更新参数，不过因为这里使用的数据是随机选择的 mini batch 数据，所以又称为随机梯度下降法 （stochastic gradient descent）。随机梯度下降法一般由一个名为 SGD 的函来实现。

基于测试数据的评价

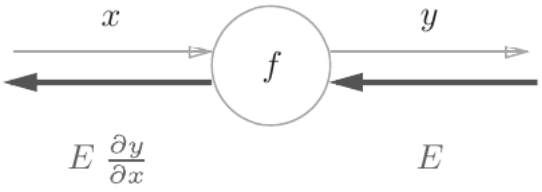
神经网络的学习中，必须确认是否能够正确识别训练数据以外的其他数据，即确认是否会发生过拟合。过拟合是指，虽然训练数据中的数字图像能被正确辨别，但是不在训练数据中的数字图像却无法被识别的现象。

下面的代码在进行学习的过程中，会定期地对训练数据和测试数据记录识别精度。这里，每经过一个epoch，我们都会记录下训练数据和测试数据的识别精度。如果两个数据间的识别精度相差不大，则说明没有过拟合。

epoch 是一个单位。一个 epoch 表示学习中所有训练数据均被使用过一次时的更新次数。比如，对于 10000 笔训练数据，用大小为 100 笔数据的 mini-batch 进行学习时，重复随机梯度下降法 100 次，所有的训练数据就都被“看过”了 。此时，100 次就是一个 epoch。

误差反向传播法

传递这个局部导数的原理，是基于链式法则 （chain rule）。

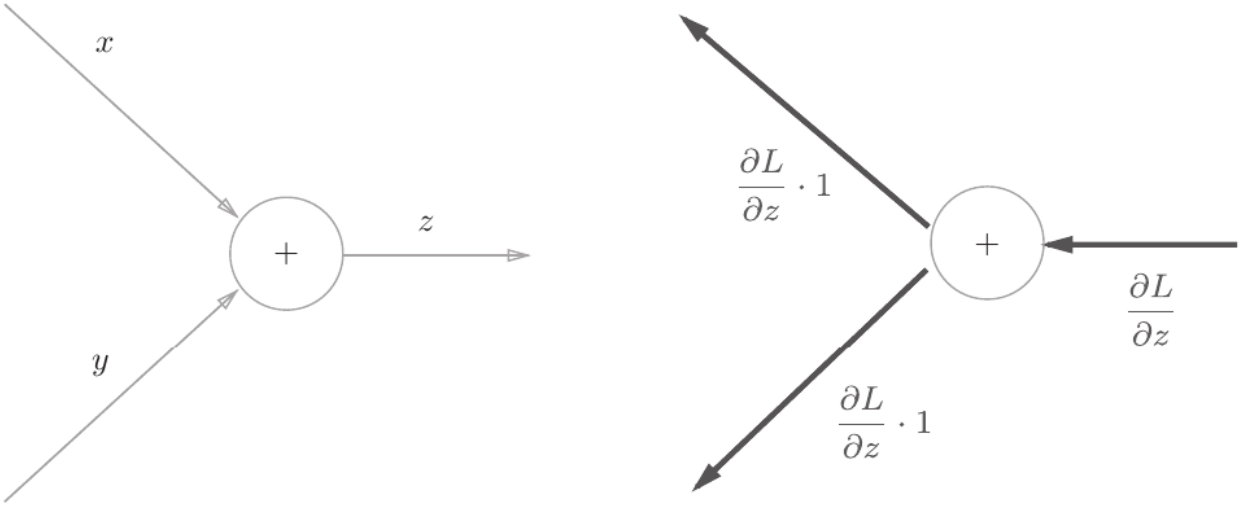


反向传播

本节将以“+”和“×”等运算为例，介绍反向传播的结构。

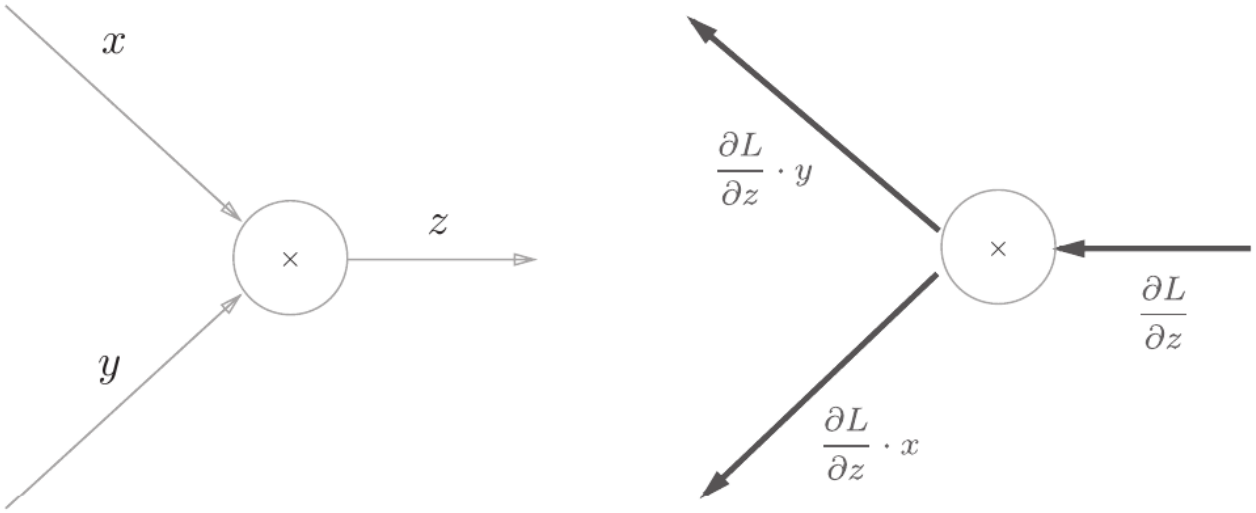
加法节点的反向传播

因为加法节点的反向传播只乘以 1，所以输入的值会原封不动地流向下一个节点。



乘法节点的反向传播

乘法的反向传播会将上游的值乘以正向传播时的输入信号的“翻转值”后传递给下游。



简单层的实现

乘法层的实现

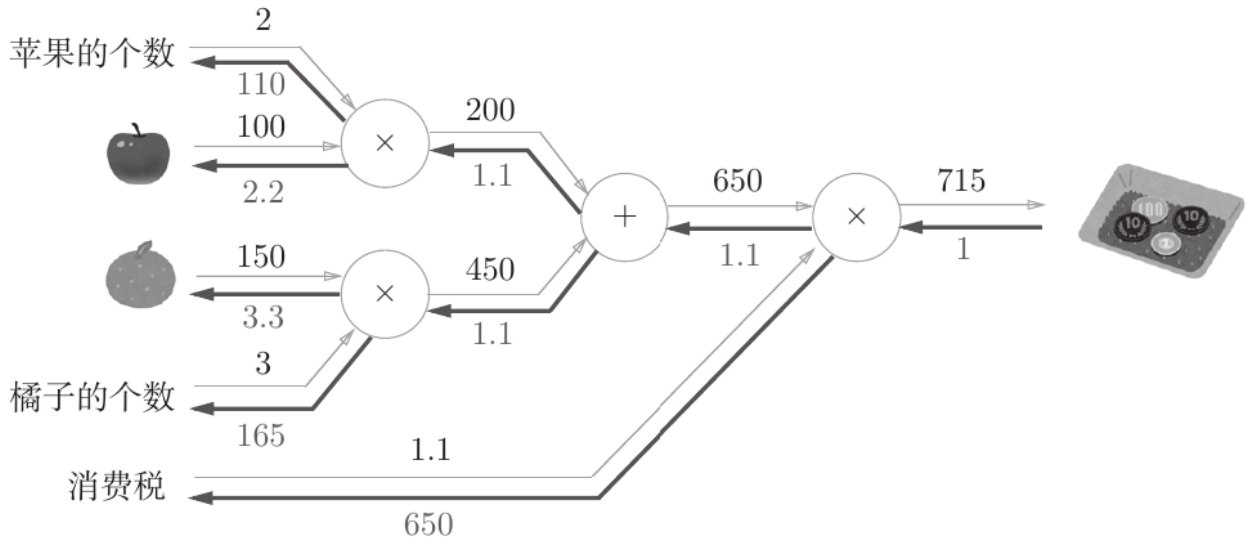
层的实现中有两个共通的方法（接口） forward() 和 backward()。forward()对应正向传播，backward() 对应反向传播。

```python
class MulLayer:
    def __init__(self):
        self.x = None
        self.y = None
    def forward(self, x, y):
        self.x = x
        self.y = y
        out = x * y
        return out
    def backward(self, dout):
        dx = dout * self.y # 翻转 x 和 y
        dy = dout * self.x
        return dx, dy
```

加法层的实现

```python
class AddLayer:
    def __init__(self):
        pass
    def forward(self, x, y):
        out = x + y
        return out
    def backward(self, dout):
        dx = dout * 1
        dy = dout * 1
        return dx, dy
```

用Python实现图中的计算图



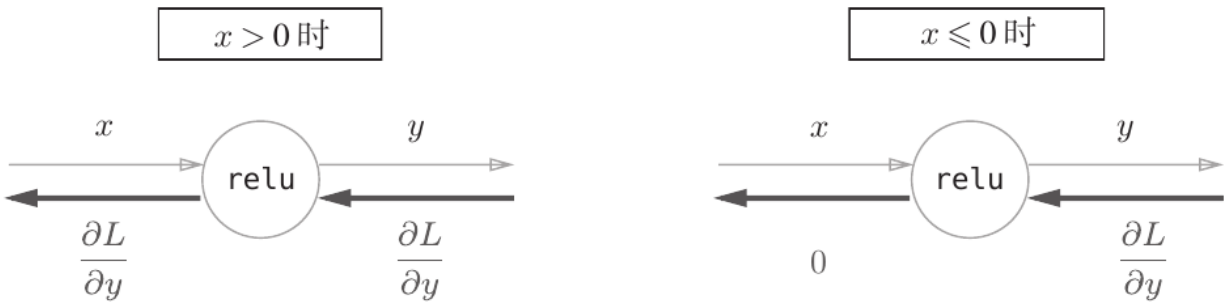

```python
apple = 100
apple_num = 2
orange = 150
orange_num = 3
tax = 1.1

# layer
mul_apple_layer = MulLayer()
mul_orange_layer = MulLayer()
add_apple_orange_layer = AddLayer()
mul_tax_layer = MulLayer()

# forward
apple_price = mul_apple_layer.forward(apple, apple_num) #(1)
```

```
orange_price = mul_orange_layer.forward(orange, orange_num) #(2)
all_price = add_apple_orange_layer.forward(apple_price,
orange_price) #(3)
price = mul_tax_layer.forward(all_price, tax) #(4)

# backward
dprice = 1
dall_price, dtax = mul_tax_layer.backward(dprice) #(4)
dapple_price, dorange_price =
add_apple_orange_layer.backward(dall_price) #(3)
dorange, dorange_num = mul_orange_layer.backward(dorange_price)
#(2)
dapple, dapple_num = mul_apple_layer.backward(dapple_price) #(1)

print(price) # 715
print(dapple_num, dapple, dorange, dorange_num, dtax) # 110 2.2
3.3 165 650
```

激活函数层的实现

这里，我们把构成神经网络的层实现为一个类。先来实现激活函数的 ReLU 层和 Sigmoid 层。

ReLU 层

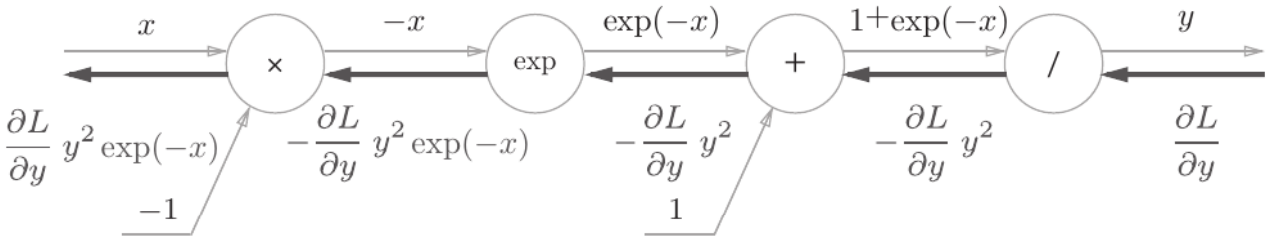


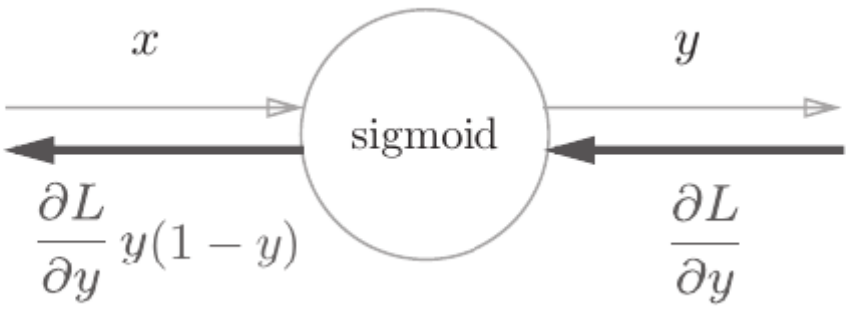
```python
class Relu:
    def __init__(self):
        self.mask = None
    def forward(self, x):
        self.mask = (x <= 0)
        out = x.copy()
        out[self.mask] = 0
        return out
    def backward(self, dout):
        dout[self.mask] = 0
        dx = dout
        return dx
```

ReLU 类有实例变量 mask 。这个变量 mask 是由 True/False 构成的 NumPy 数组，它会把正向传播时的输入 x 的元素中小于等于 0 的地方保存为 True ，其他地方（大于 0 的元素）保存为 False。

ReLU 层的作用就像电路中的开关一样。正向传播时，有电流通过的话，就将开关设为 ON；没有电流通过的话，就将开关设为 OFF。反向传播时，开关为 ON 的话，电流会直接通过；开关为 OFF 的话，则不会有电流通过。

Sigmoid 层





```python
class Sigmoid:
    def __init__(self):
        self.out = None
    def forward(self, x):
        out = 1 / (1 + np.exp(-x))
        self.out = out
        return out
    def backward(self, dout):
        dx = dout * (1.0 - self.out) * self.out
        return dx
```

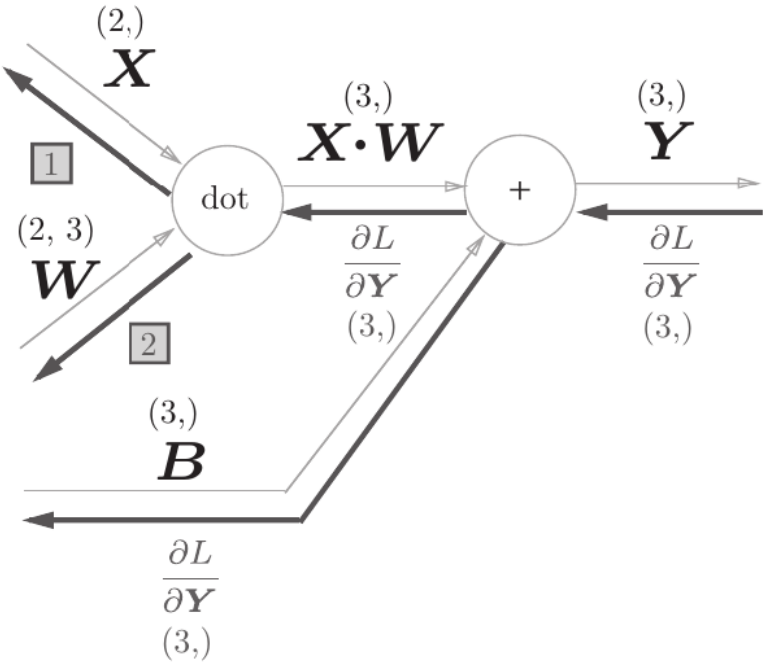
Affine/Softmax 层的实现

Affine 层

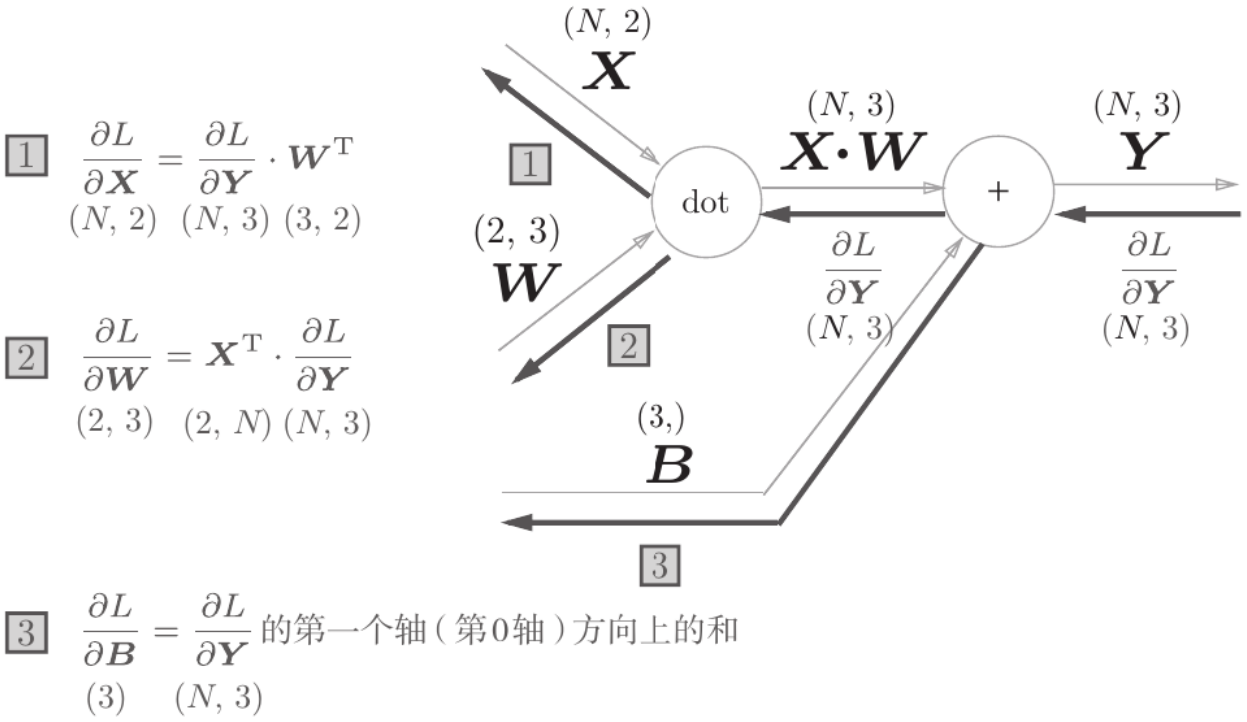
神经网络的正向传播中进行的矩阵的乘积运算在几何学领域被称为“仿射变换”。因此，这里将进行仿射变换的处理实为“Affine 层”。

$$\boxed{1}\quad \underset{(2,)}{\frac{\partial L}{\partial \boldsymbol{X}}} = \underset{(3,)}{\frac{\partial L}{\partial \boldsymbol{Y}}} \cdot \underset{(3,\ 2)}{\boldsymbol{W}^{\mathrm{T}}}$$

$$\boxed{2}\quad \underset{(2,\ 3)}{\frac{\partial L}{\partial \boldsymbol{W}}} = \underset{(2,\ 1)}{\boldsymbol{X}^{\mathrm{T}}} \cdot \underset{(1,\ 3)}{\frac{\partial L}{\partial \boldsymbol{Y}}}$$



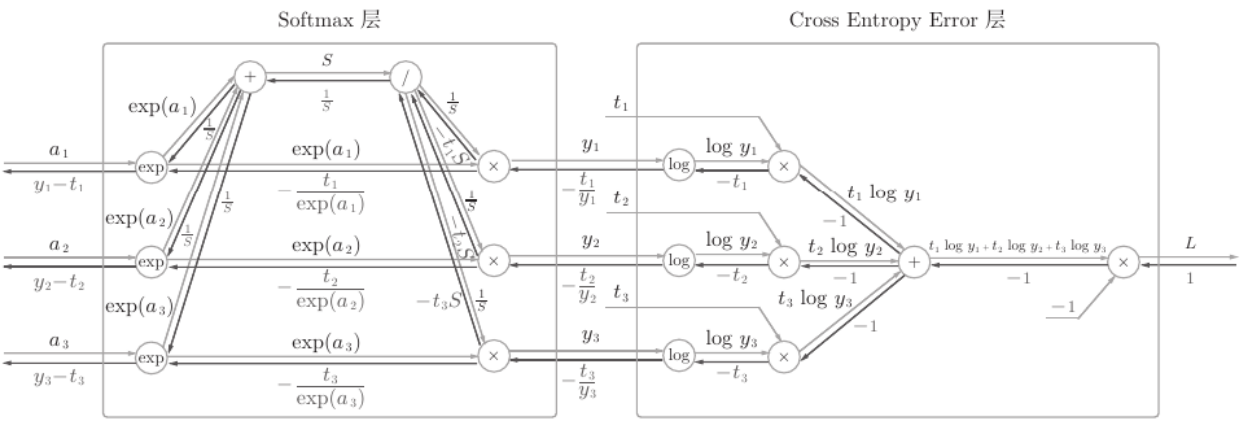
如果是批版本的处理，计算图如下：

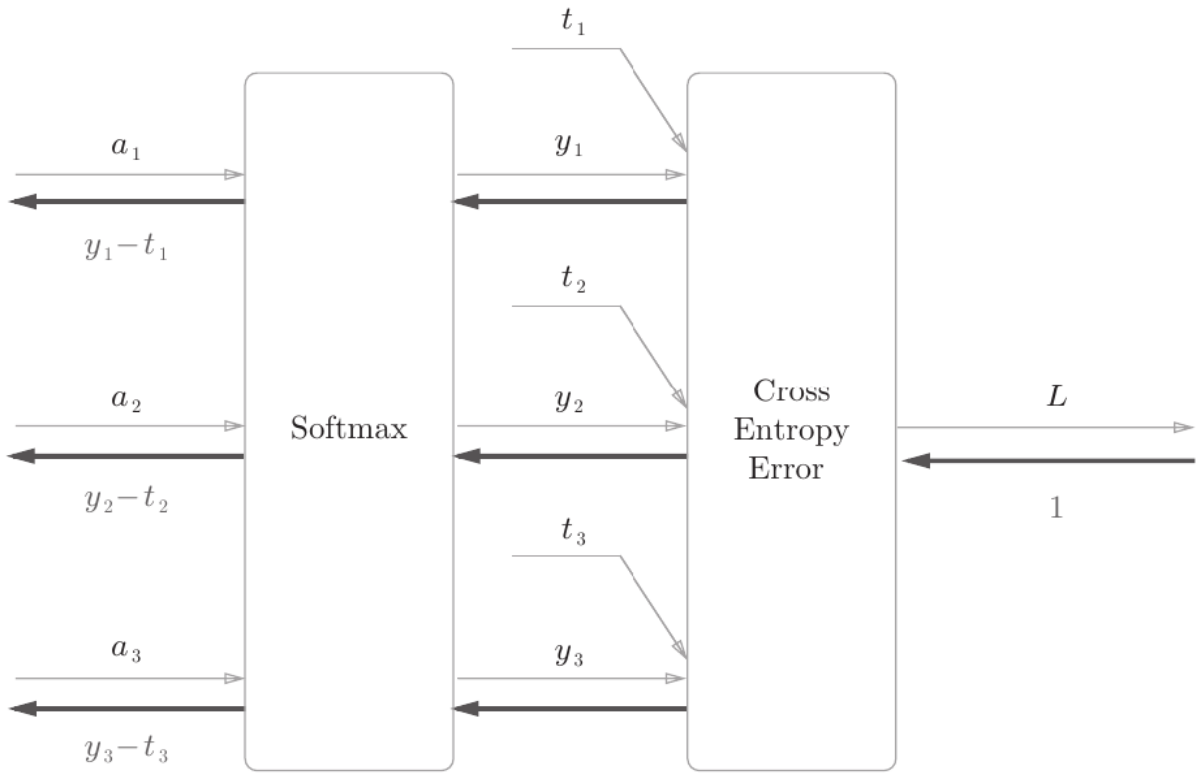



```python
class Affine:
    def __init__(self, W, b):
    self.W = W
    self.b = b
    self.x = None
    self.dW = None
    self.db = None
    def forward(self, x):
        self.x = x
        out = np.dot(x, self.W) + self.b
        return out
    def backward(self, dout):
        dx = np.dot(dout, self.W.T)
        self.dW = np.dot(self.x.T, dout)
        self.db = np.sum(dout, axis=0)
        return dx
```

Softmax-with-Loss 层

softmax 函数称为 softmax 层，交叉熵误差称为 Cross Entropy Error 层，两者的组合称为 Softmax-with-Loss 层。





Softmax 层的反向传播得到了（y1 − t1, y2 − t2, y3 − t3 ）这样“漂亮”的结果。由于（y1, y2, y3 ）是 Softmax 层的输出，（t1, t2, t3 ）是监督数据，所以（y1 − t1, y2 − t2, y3 − t3 ）是 Softmax 层的输出和教师标签的差分。神经网络的反向传播会把这个差分表示的误差传递给前面的层，这是神经网络学习中的重要性质。

```python
class SoftmaxWithLoss:
    def __init__(self):
        self.loss = None # 损失
        self.y = None
         # softmax 的输出
        self.t = None
     # 监督数据（one-hot vector）
    def forward(self, x, t):
        self.t = t
        self.y = softmax(x)
```

```
        self.loss = cross_entropy_error(self.y, self.t)
        return self.loss
    def backward(self, dout=1):
        batch_size = self.t.shape[0]
        dx = (self.y - self.t) / batch_size
        return dx
```

误差反向传播法的实现

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from common.layers import *
from common.gradient import numerical_gradient
from collections import OrderedDict
class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size,
        weight_init_std=0.01):
        # 初始化权重
        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)
        # 生成层
        self.layers = OrderedDict()
        self.layers['Affine1'] = \
        Affine(self.params['W1'], self.params['b1'])
        self.layers['Relu1'] = Relu()
        self.layers['Affine2'] = \
        Affine(self.params['W2'], self.params['b2'])
        self.lastLayer = SoftmaxWithLoss()
    def predict(self, x):
        for layer in self.layers.values():
            x = layer.forward(x)
        return x
    # x: 输入数据 , t: 监督数据
    def loss(self, x, t):
        y = self.predict(x)
```

```
        return self.lastLayer.forward(y, t)
    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        if t.ndim != 1 : t = np.argmax(t, axis=1)
            accuracy = np.sum(y == t) / float(x.shape[0])
        return accuracy
    # x: 输入数据 , t: 监督数据
    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):
        # forward
        self.loss(x, t)
        # backward
        dout = 1
        dout = self.lastLayer.backward(dout)
        layers = list(self.layers.values())
        layers.reverse()
        for layer in layers:
            dout = layer.backward(dout)
        # 设定
        grads = {}
        grads['W1'] = self.layers['Affine1'].dW
        grads['b1'] = self.layers['Affine1'].db
        grads['W2'] = self.layers['Affine2'].dW
        grads['b2'] = self.layers['Affine2'].db
        return grads
```

OrderedDict 是有序字典，“有序”是指它可以记住向字典里添加元素的顺序。因此，神经网络的正向传播只需按照添加元素的顺序调用各层的 forward() 方法就可以完成处理，而反向传播只需要按照相反的顺序调用各层即可。

像这样通过将神经网络的组成元素以层的方式实现，可以轻松地构建神经网络。这个用层进行模块化的实现具有很大优点。因为想另外构建一个神经网络（比如 5 层、10 层、20 层……的大的神经网络）时，只需像组装乐高积木那样添加必要的层就可以了。之后，通过各个层内部实现的正向传播和反向传播，就可以正确计算进行识别处理或学习所需的梯度。

误差反向传播法的梯度确认

到目前为止，我们介绍了两种求梯度的方法。一种是基于数值微分的方法，另一种是解析性地求解数学式的方法。后一种方法通过使用误差反向传播法，即使存在大量的参数，也可以高效地计算梯度。

确认数值微分求出的梯度结果和误差反向传播法求出的结果是否一致（严格地讲，是非常相近）的操作称为梯度确认 （gradient check）。梯度确认的代码实现如下所示：

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet
# 读入数据
(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)
x_batch = x_train[:3]
t_batch = t_train[:3]
grad_numerical = network.numerical_gradient(x_batch, t_batch)
grad_backprop = network.gradient(x_batch, t_batch)
# 求各个权重的绝对误差的平均值
for key in grad_numerical.keys():
    diff = np.average( np.abs(grad_backprop[key] -
grad_numerical[key]) )
    print(key + ":" + str(diff))
```

使用误差反向传播法的学习

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
```

```
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet
# 读入数据
(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)
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]
    # 通过误差反向传播法求梯度
    grad = network.gradient(x_batch, t_batch)
    # 更新
    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)
    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)
```

与学习相关的技巧

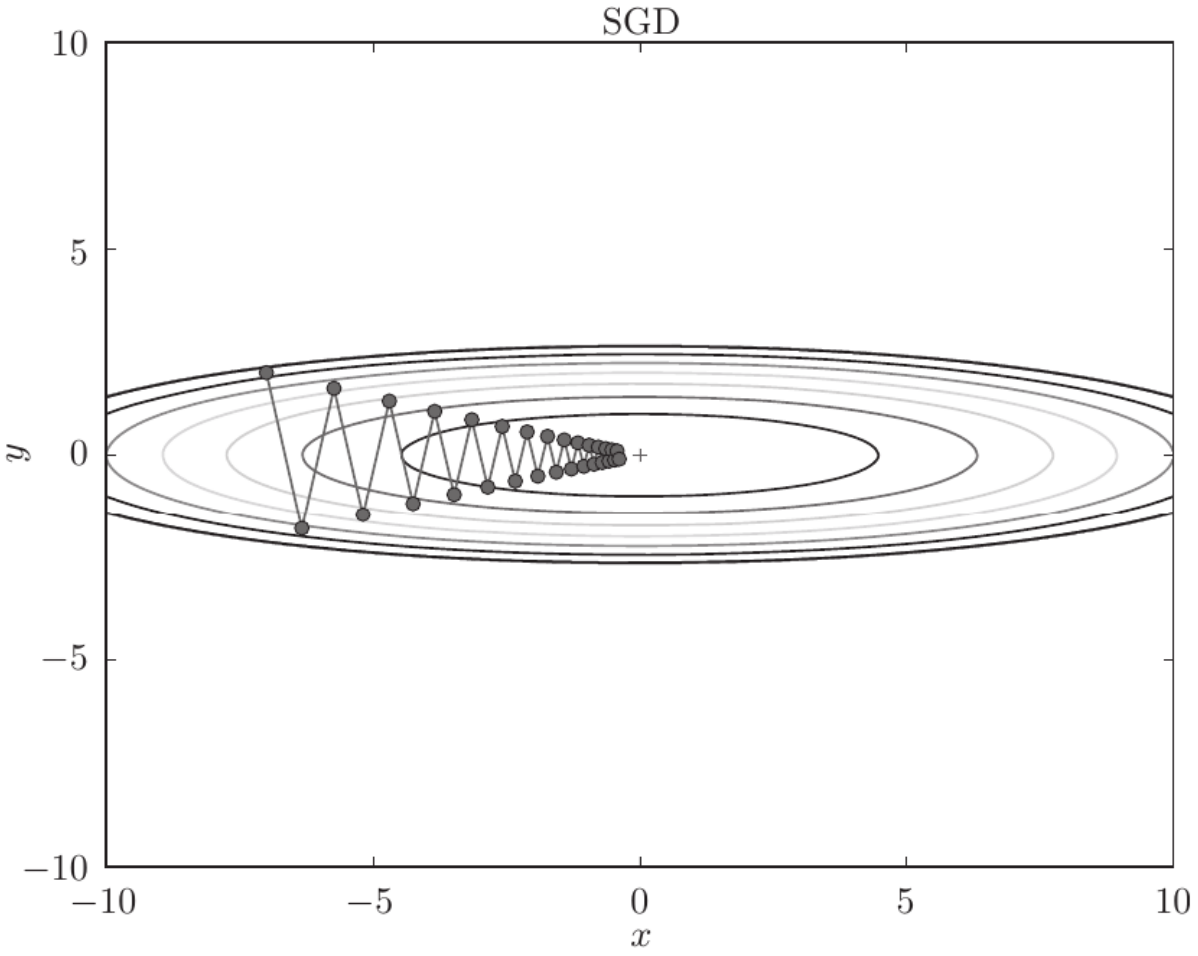
参数的更新

神经网络的学习的目的是找到使损失函数的值尽可能小的参数。这是寻找最优参数的问题，解决这个问题的过程称为最优化 （optimization）。

SGD的缺点

SGD 的缺点是，如果函数的形状非均向（anisotropic），比如呈延伸状，搜索的路径就会非常低效。因此，我们需要比单纯朝梯度方向前进的 SGD 更聪-明的方法。SGD 低效的根本原因是，梯度的方向并没有指向最小值的方向。

为了改正SGD的缺点，下面我们将介绍Momentum、AdaGrad、Adam这3种方法来取代SGD。



Momentum

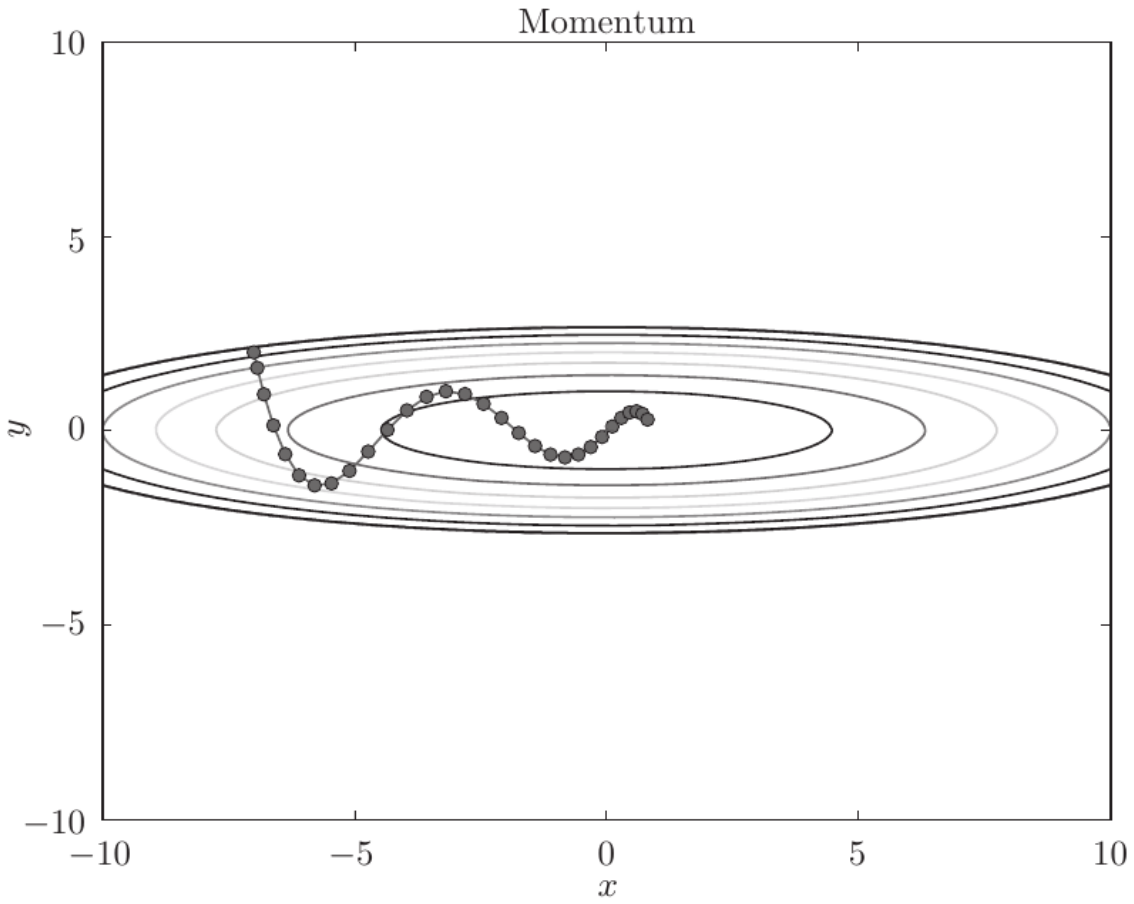
Momentum 是“动量”的意思，和物理有关。用数学式表示 Momentum 方法，如下所示。

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

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

这里新出现了一个变量 v，对应物理上的速度。式表示了物体在梯度方向上受力，在这个力的作用下，物体的速度增加这一物理法则。在物体不受任何力时，该项承担使物体逐渐减速的任务（α 设定为 0.9 之类的值），对应物理上的地面摩擦或空气阻力。

```python
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]
```



AdaGrad

在神经网络的学习中，学习率（数学式中记为 η）的值很重要。学习率过小，会导致学习花费过多时间；反过来，学习率过大，则会导致学习发散而不能正确进行。

在关于学习率的有效技巧中，有一种被称为学习率衰减 （learning ratedecay）的方法，即随着学习的进行，使学习率逐渐减小。实际上，一开始“多”学，然后逐渐“少”学的方法，在神经网络的学习中经常被使用。

AdaGrad 会为参数的每个元素适当地调整学习率，与此同时进行学习（AdaGrad 的 Ada 来自英文单词 Adaptive，即“适当的”的意思）。下面，让我们用数学式表示 AdaGrad 的更新方法。

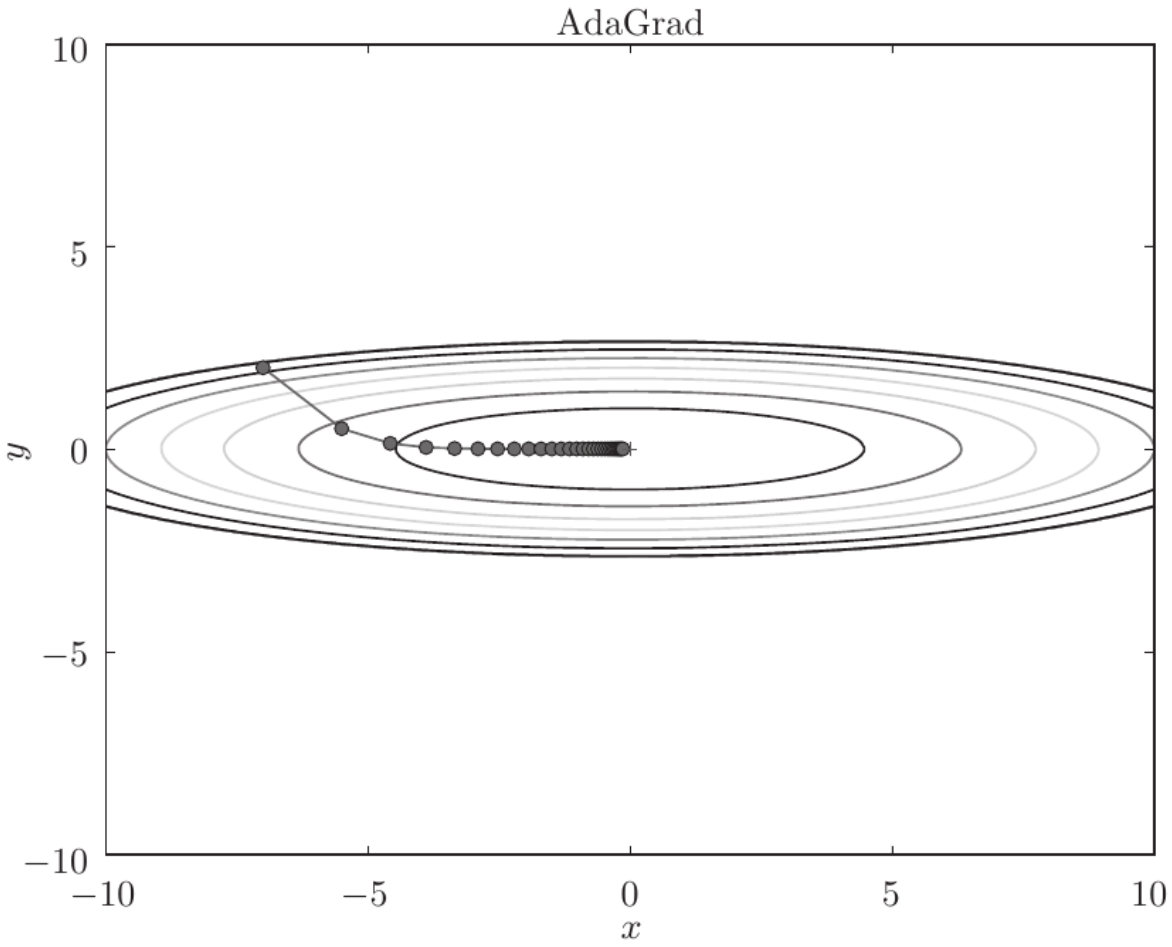
$$\boldsymbol{h} \leftarrow \boldsymbol{h} + \frac{\partial L}{\partial \boldsymbol{W}} \odot \frac{\partial L}{\partial \boldsymbol{W}}$$

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

它保存了以前的所有梯度值的平方和。然后，在更新参数时，通过乘以 ，就可以调整学习的尺度。这意味着，参数的元素中变动较大（被大幅更新）的元素的学习率将变小。也就是说，可以按参数的元素进行学习率衰减，使变动大的参数的学习率逐渐减小。

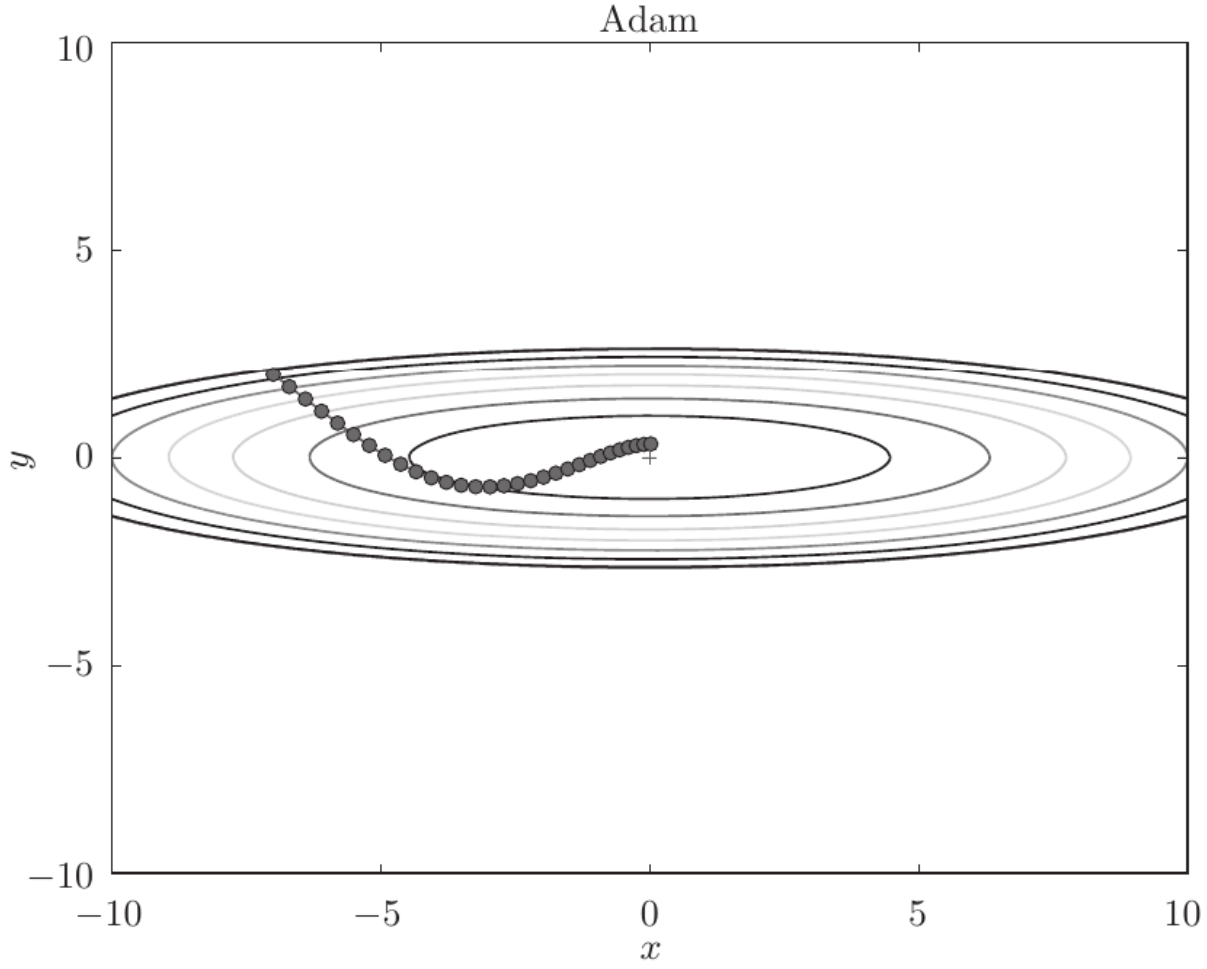
AdaGrad 会记录过去所有梯度的平方和。因此，学习越深入，更新的幅度就越小。实际上，如果无止境地学习，更新量就会变为0，完全不再更新。为了改善这个问题，可以使用RMSProp方法。RMSProp 方法并不是将过去所有的梯度一视同仁地相加，而是逐渐地遗忘过去的梯度，在做加法运算时将新梯度的信息更多地反映出来。这种操作从专业上讲，称为“指数移动平均”，呈指数函数式地减小过去的梯度的尺度。

```
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)
```



Adam

Adam 是 2015 年提出的新方法。它的理论有些复杂，直观地讲，就是融合了 Momentum 和 AdaGrad 的方法。通过组合前面两个方法的优点，有望实现参数空间的高效搜索。此外，进行超参数的“偏置校正”也是 Adam 的特征。



Adam 会设置 3 个超参数。一个是学习率（论文中以 α 出现），另外两个是一次 momentum 系数 β 1 和二次 momentum 系数 β 2 。根据论文，标准的设定值是 β 1 为 0.9，β 2 为 0.999。设置了这些值后，大多数情况下都能顺利运行。

权重的初始值

可以将权重初始值设为 0 吗

后面我们会介绍抑制过拟合、提高泛化能力的技巧——权值衰减（weightdecay）。简单地说，权值衰减就是一种以减小权重参数的值为目的进行学习的方法。通过减小权重参数的值来抑制过拟合的发生。

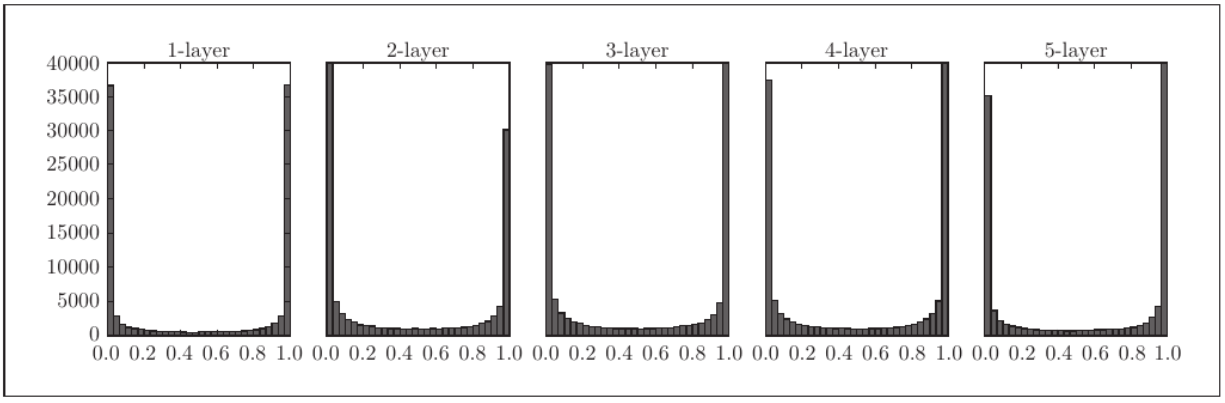
如果想减小权重的值，一开始就将初始值设为较小的值才是正途。实际上，在这之前的权重初始值都是像 0.01 * np.random.randn(10, 100) 这样，使用由高斯分布生成的值乘以 0.01 后得到的值（标准差为 0.01 的高斯分布）。

高斯分布在连续函数中熵最大。

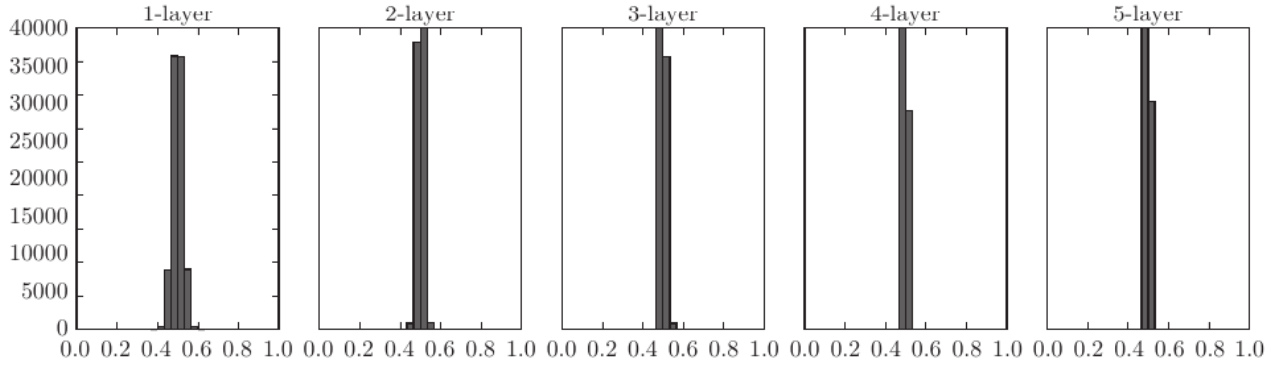
将权重初始值设为0 的话，将无法正确进行学习。严格地说，为什么不能将权重初始值设成一样的值呢？这是因为在误差反向传播法中，所有的权重值都会进行相同的更新。比如，在 2 层神经网络中，假设第 1 层和第 2 层的权重为 0。这样一来，正向传播时，因为输入层的权重为 0，所以第 2 层的神经元全部会被传递相同的值。第 2 层的神经元中全部输入相同的值，这意味着反向传播时第 2 层的权重全部都会进行相同的更新。因此，权重被更新为相同的值，并拥有了对称的（重复的值）。这使得神经网络拥有许多不同的权重的意义丧失了。为了防止“权重均一化”（严格地讲，是为了瓦解权重的对称结构），必须随机生成初始值。

隐藏层的激活值的分布

偏向 0 和 1 的数据分布（sigmoid函数等）会造成反向传播中梯度的值不断变小，最后消失。这个问题称为梯度消失 （gradient vanishing）。层次加深的深度学习中，梯度消失的问题可能会更加严重。



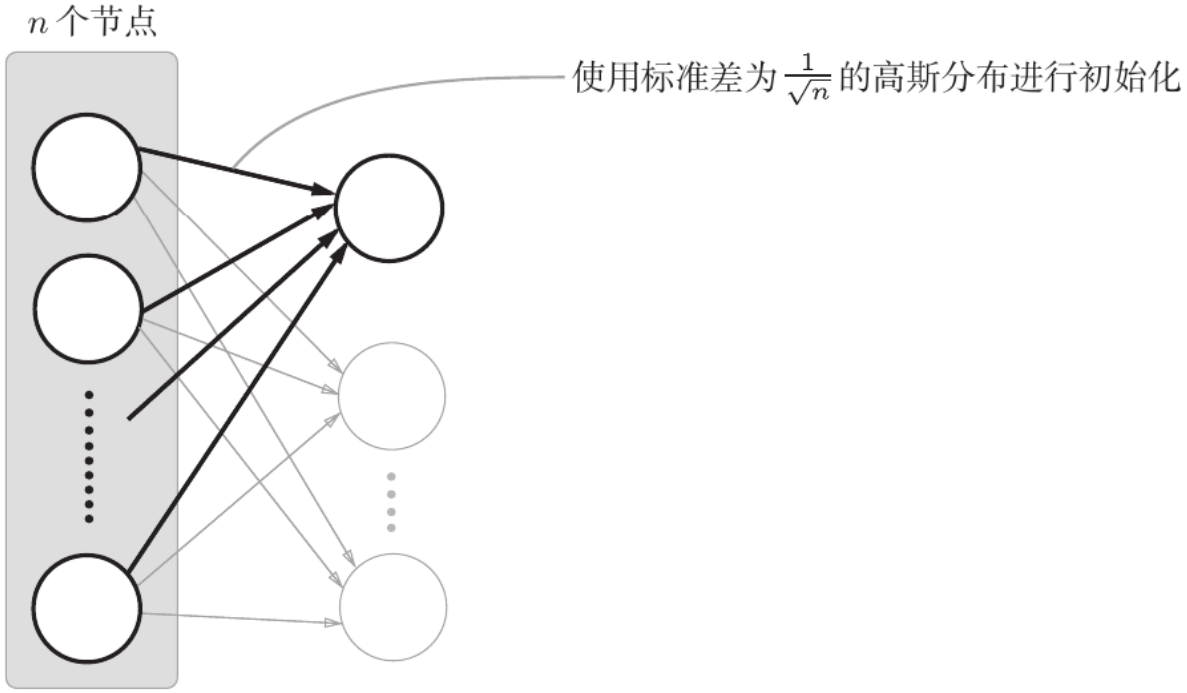
使用标准差为 0.01 的高斯分布时，各层的激活值的分布如图

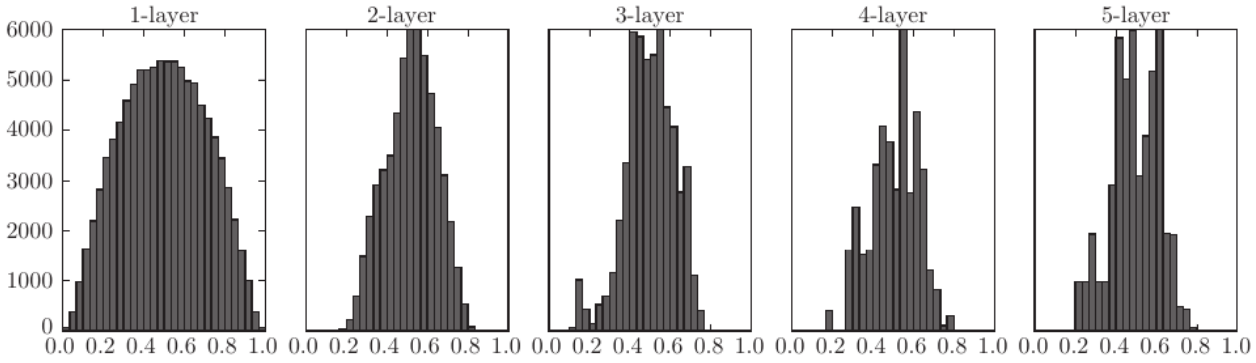


但是，激活值的分布有所偏向，说明在表现力上会有很大问题。为什么这么说呢？因为如果有多个神经元都输出几乎相同的值，那它们就没有存在的意义了。比如，如果 100 个神经元都输出几乎相同的值，那么也可以由 1 个神经元来表达基本相同的事情。因此，激活值在分布上有所偏向会出现“表现力受限”的问题。

各层的激活值的分布都要求有适当的广度。为什么呢？因为通过在各层间传递多样性的数据，神经网络可以进行高效的学习。反过来，如果传递的是有所偏向的数据，就会出现梯度消失或者“表现力受限”的问题，导致学习可能无法顺利进行。

接着，我们尝试使用 Xavier Glorot 等人推荐的权重初始值（俗称“Xavier 初始值”）。Xavier 的论文中，为了使各层的激活值呈现出具有相同广度的分布，推导了合适的权重尺度。



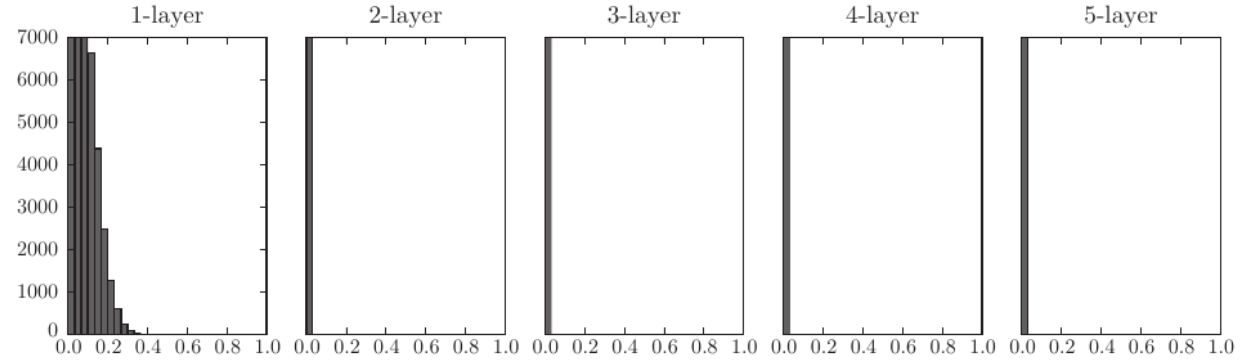


分布中，后面的层的分布呈稍微歪斜的形状。如果用 tanh 函数（双曲线函数）代替 sigmoid 函数，这个稍微歪斜的问题就能得到改善。实际上，使用 tanh 函数后，会呈漂亮的吊钟型分布。tanh 函数和 sigmoid 函数同是 S 型曲线函数，但 tanh 函数是关于原点 (0, 0) 对称的 S 型曲线，而 sigmoid 函数是关于 (x, y)=(0, 0.5) 对称的 S 型曲线。众所周知，用作激活函数的函数最好具有关于原点对称的性质。

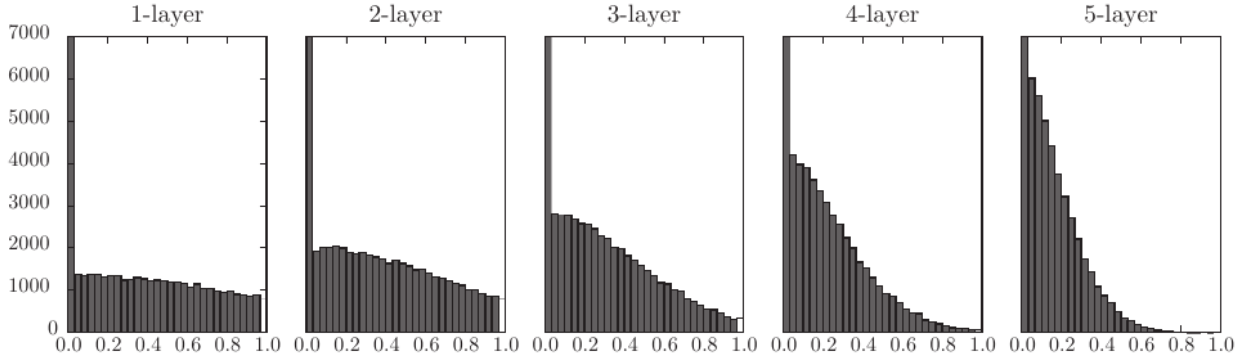
ReLU 的权重初始值

Xavier 初始值是以激活函数是线性函数为前提而推导出来的。因为sigmoid 函数和 tanh 函数左右对称，且中央附近可以视作线性函数，所以适合使用 Xavier 初始值。但当激活函数使用 ReLU 时，一般推荐使用 ReLU 专用的初始值，也就是 Kaiming He 等人推荐的初始值，也称为“He 初始值”。

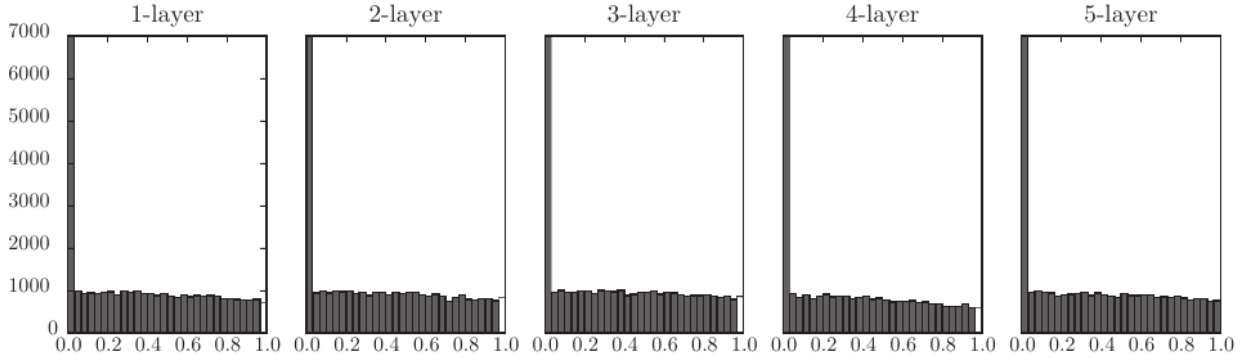
当前一层的节点数为 n 时，He 初始值使用标准差为根号下n分之2的高斯分布。当Xavier 初始值是根号下n分之一时，（直观上）可以解释为，因为 ReLU 的负值区域的值为 0，为了使它更有广度，所以需要 2 倍的系数。



权重初始值为标准差是 0.01 的高斯分布时



权重初始值为 Xavier 初始值时



权重初始值为 He 初始值时

总结一下，当激活函数使用 ReLU 时，权重初始值使用 He 初始值，当激活函数为 sigmoid 或 tanh 等 S 型曲线函数时，初始值使用 Xavier 初始值。这是目前的最佳实践。

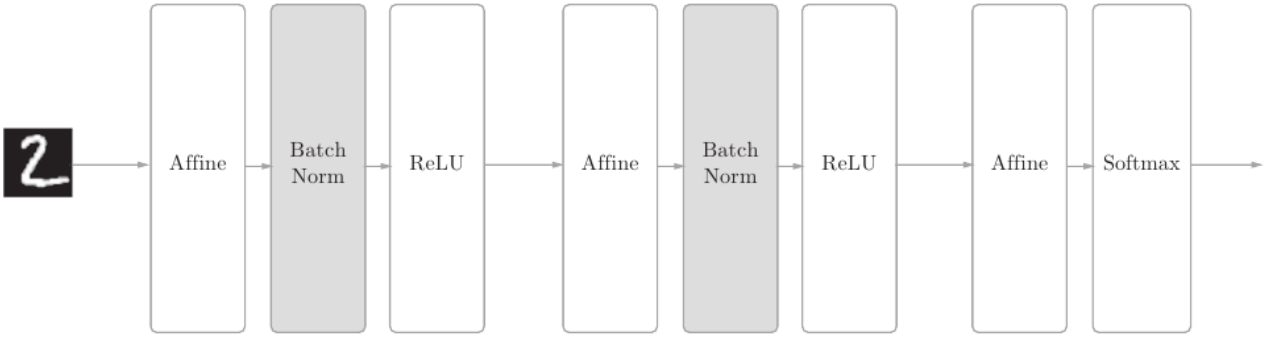
Batch Normalization

在上一节，我们观察了各层的激活值分布，并从中了解到如果设定了合适的权重初始值，则各层的激活值分布会有适当的广度，从而可以顺利地进行学习。那么，为了使各层拥有适当的广度，“强制性”地调整激活值的分布会怎样呢？实际上，Batch Normalization方法就是基于这个想法而产生的。

Batch Norm 有以下优点:

- 可以使学习快速进行（可以增大学习率）。
- 不那么依赖初始值（对于初始值不用那么神经质）。
- 抑制过拟合（降低 Dropout 等的必要性）。

如前所述，Batch Norm 的思路是调整各层的激活值分布使其拥有适当的广度。为此，要向神经网络中插入对数据分布进行正规化的层，即 BatchNormalization 层（下文简称 Batch Norm 层）。



Batch Norm，顾名思义，以进行学习时的 mini-batch 为单位，按 mini-batch 进行正规化。具体而言，就是进行使数据分布的均值为 0、方差为 1 的正规化。用数学式表示的话，如下所示。

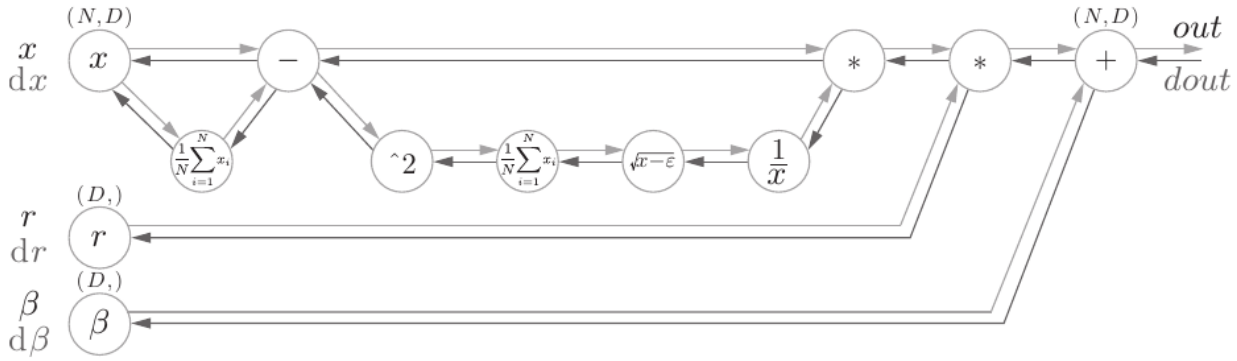
$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

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

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

接着，Batch Norm 层会对正规化后的数据进行缩放和平移的变换，用数学式可以如下表示。

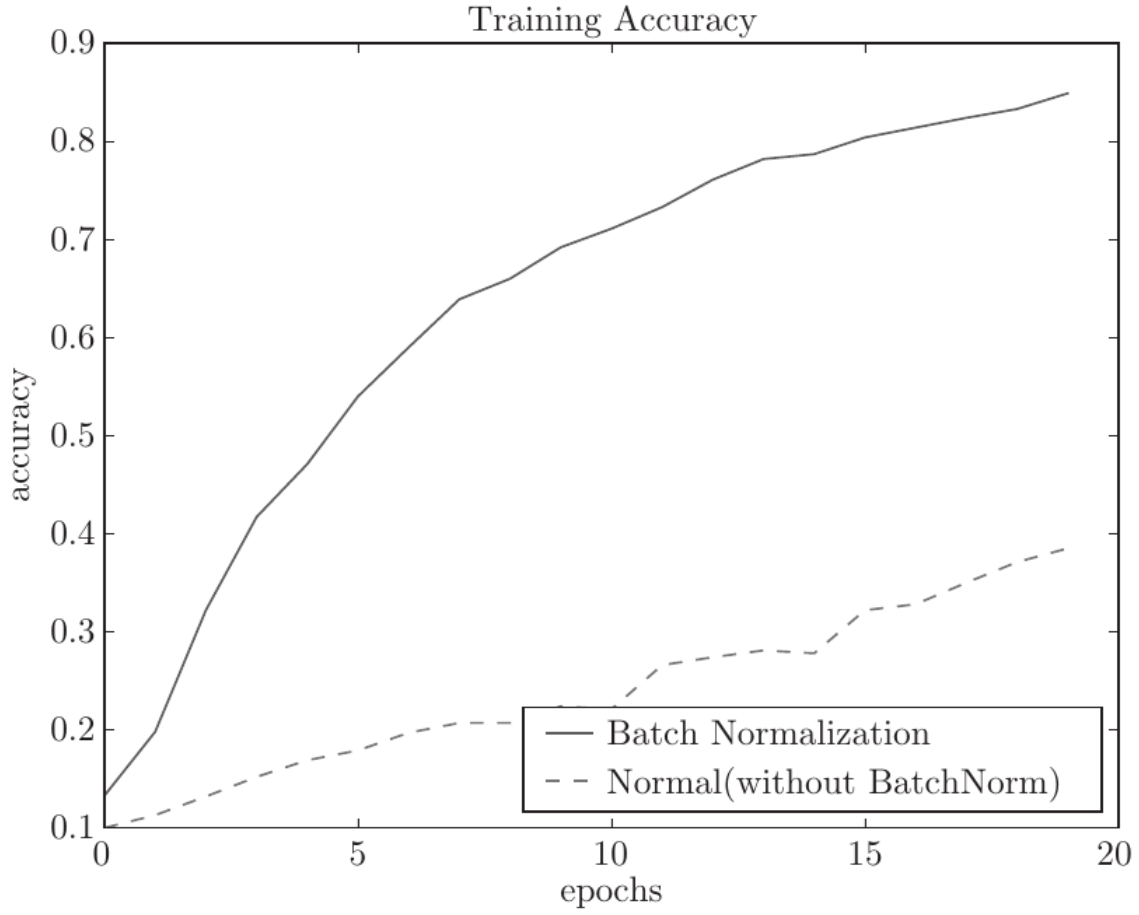
$$y_i \leftarrow \gamma \hat{x}_i + \beta$$



Frederik Kratzert 的 博 客“Understanding the backward pass through Batch Normalization Layer” 里有详细说明，感兴趣的读者可以参考一下。

Batch Normalization 的评估

现在我们使用 Batch Norm 层进行实验。首先，使用 MNIST 数据集，观察使用 Batch Norm 层和不使用 Batch Norm 层时学习的过程会如何变化.



使用 Batch Norm 后，学习进行得更快了。我们发现，几乎所有的情况下都是使用 Batch Norm 时学习进行得更快。同时也可以发现，实际上，在不使用 Batch Norm 的情况下，如果不赋予一个尺度好的初始值，学习将完全无法进行。

综上，通过使用 Batch Norm，可以推动学习的进行。并且，对权重初始值变得健壮（“对初始值健壮”表示不那么依赖初始值）。

正则化

机器学习的问题中，过拟合是一个很常见的问题。过拟合指的是只能拟合训练数据，但不能很好地拟合不包含在训练数据中的其他数据的状态。机器学习的目标是提高泛化能力，即便是没有包含在训练数据里的未观测数据，也希望模型可以进行正确的识别。我们可以制作复杂的、表现力强的模型，

发生过拟合的原因，主要有以下两个:

- 模型拥有大量参数、表现力强。
- 训练数据少。

权值衰减

权值衰减是一直以来经常被使用的一种抑制过拟合的方法。该方法通过在学习的过程中对大的权重进行惩罚，来抑制过拟合。很多过拟合原本就是因为权重参数取值过大才发生的。

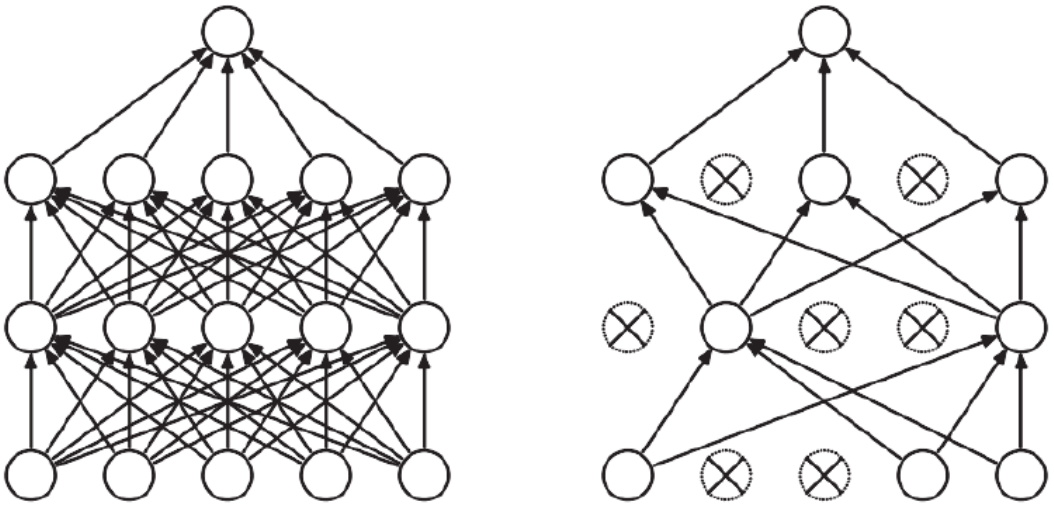
例如，为损失函数加上权重的平方范数（L2 范数）。这样一来，就可以抑制权重变大。用符号表示的话，如果将权重记为 W，L2 范数的权值衰减就是 0.5*λW^2 ，然后将这个 加到损失函数上。这里，λ 是控制正则化强度的超参数。λ设置得越大，对大的权重施加的惩罚就越重。此外， 开头的是用于将的求导结果变成 λW 的调整用常量。

在求权重梯度的计算中，要为之前的误差反向传播法的结果加上正则化项的导数λW。

Dropout

如果网络的模型变得很复杂，只用权值衰减就难以应对了。在这种情况下，我们经常会使用 Dropout方法。

Dropout 是一种在学习的过程中随机删除神经元的方法。训练时，随机选出隐藏层的神经元，然后将其删除。被删除的神经元不再进行信号的传递。



(a) Standard Neural Net　　(b) After applying dropout.

```python
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
```

正向传播时传递了信号的神经元，反向传播时按原样传递信号；正向传播时，没有传递信号的神经元，反向传播时信号将停在那里。

机器学习中经常使用集成学习。所谓集成学习，就是让多个模型单独进行学习，推理时再取多个模型的输出的平均值。用神经网络的语境来说，比如，准备 5 个结构相同（或者类似）的网络，分别进行学习，测试时，以这 5 个网络的输出的平均值作为答案。实验告诉我们，通过进行集成学习，神经网络的识别精度可以提高好几个百分点。这个集成学习与 Dropout 有密切的关系。这是因为可以将 Dropout 理解为，通过在学习过程中随机删除神经元，从而每一次都让不同的模型进行学习。并且，推理时，通过对神经元的输出乘以删除比例（比如，0.5 等），可以取得模型的平均值。也就是说，可以理解成，Dropout 将集成学习的效果（模拟地）通过一个网络实现了。

超参数的验证

神经网络中，除了权重和偏置等参数，超参数 （hyper-parameter）也经常出现。这里所说的超参数是指，比如各层的神经元数量、batch 大小、参数更新时的学习率或权值衰减等。如果这些超参数没有设置合适的值，模型的性能就会很差。

验证数据

为什么不能用测试数据评估超参数的性能呢？这是因为如果使用测试数据调整超参数，超参数的值会对测试数据发生过拟合。

因此，调整超参数时，必须使用超参数专用的确认数据。用于调整超参数的数据，一般称为验证数据 （validation data）。我们使用这个验证数据来评估超参数的好坏。

超参数的最优化

进行超参数的最优化时，逐渐缩小超参数的“好值”的存在范围非常重要。所谓逐渐缩小范围，是指一开始先大致设定一个范围，从这个范围中随机选出一个超参数（采样），用这个采样到的值进行识别精度的评估；然后，多次重复该操作，观察识别精度的结果，根据这个结果缩小超参数的“好值”的范围。通过重复这一操作，就可以逐渐确定超参数的合适范围。

有报告显示，在进行神经网络的超参数的最优化时，与网格搜索等有规律的搜索相比，随机采样的搜索方式效果更好。这是因为在多个超参数中，各个超参数对最终的识别精度的影响程度不同。

超参数的范围只要“大致地指定”就可以了。所谓“大致地指定”，是指以“10 的阶乘”的尺度指定范围（也表述为“用对数尺度（log scale）指定”）。

在超参数的最优化中，要注意的是深度学习需要很长时间（比如，几天或几周）。因此，在超参数的搜索中，需要尽早放弃那些不符合逻辑的超参数。于是，在超参数的最优化中，减少学习的 epoch，缩短一次评估所需的时间是一个不错的办法。

这里介绍的超参数的最优化方法是实践性的方法。不过，这个方法与其说是科学方法，倒不如说有些实践者的经验的感觉。在超参数的最优化中，如果需要更精炼的方法，可以使用贝叶斯最优化 （Bayesian optimization）。贝叶斯最优化运用以贝叶斯定理为中心的数学理论，能够更加严密、高效地进行最优化。详细内容请参考论 文“Practical Bayesian Optimization of Machine Learning Algorithms”等。

卷积神经网络

本章的主题是卷积神经网络（Convolutional Neural Network，CNN）。

首先，来看一下 CNN 的网络结构，了解 CNN 的大致框架。CNN 和之前介绍的神经网络一样，可以像乐高积木一样通过组装层来构建。不过，CNN 中新出现了卷积层（Convolution 层）和池化层（Pooling 层）。

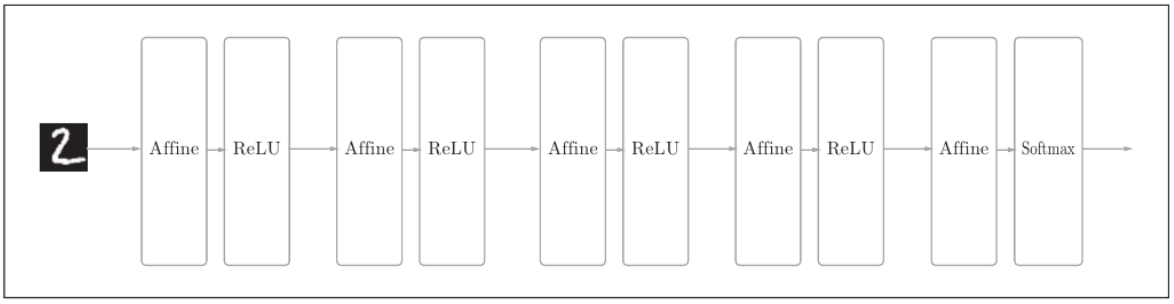


图 7-1　基于全连接层（Affine 层）的网络的例子

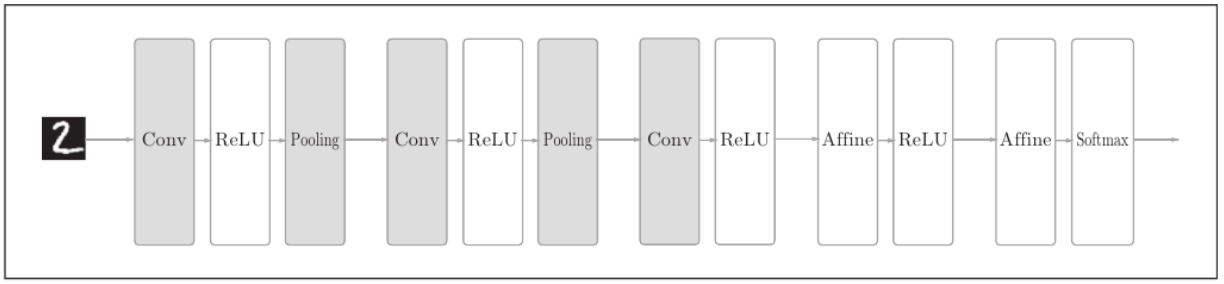


图 7-2　基于 CNN 的网络的例子：新增了 Convolution 层和 Pooling 层（用灰色的方块表示）

卷积层

全连接层存在的问题

全连接层存在什么问题呢？那就是数据的形状被“忽视”了。比如，输入数据是图像时，图像通常是高、长、通道方向上的 3 维形状。但是，向全连接层输入时，需要将 3 维数据拉平为 1 维数据。实际上，前面提到的使用了 MNIST 数据集的例子中，输入图像就是 1 通道、高 28 像素、长 28 像素的（1, 28, 28）形状，但却被排成 1 列，以 784 个数据的形式输入到最开始的Affine 层。

图像是 3 维形状，这个形状中应该含有重要的空间信息。比如，空间上邻近的像素为相似的值、RBG 的各个通道之间分别有密切的关联性、相距较远的像素之间没有什么关联等，3 维形状中可能隐藏有值得提取的本质模式。

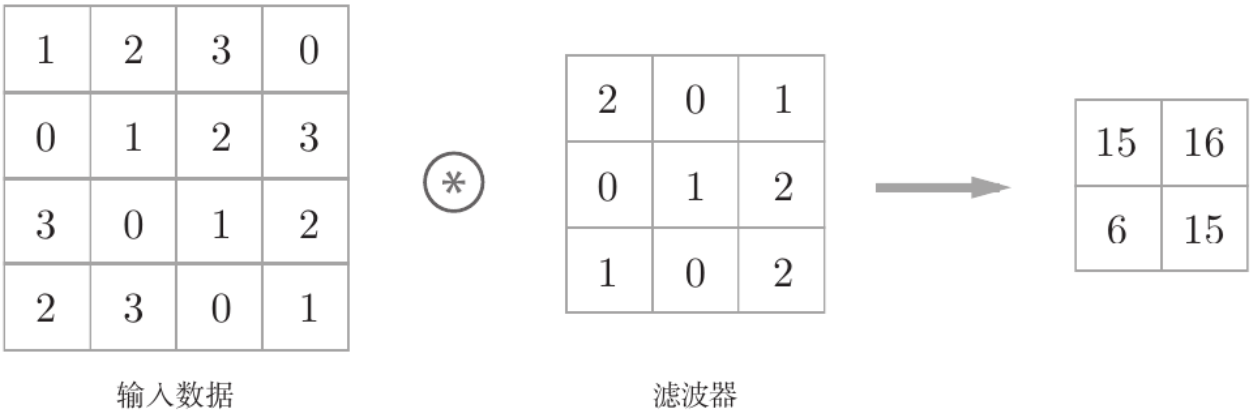
而卷积层可以保持形状不变。当输入数据是图像时，卷积层会以 3 维数据的形式接收输入数据，并同样以 3 维数据的形式输出至下一层。因此，在 CNN 中，可以（有可能）正确理解图像等具有形状的数据。

CNN 中，有时将卷积层的输入输出数据称为特征图（featuremap）。其中，卷积层的输入数据称为输入特征图（input feature map），输出数据称为输出特征图（output feature map）。

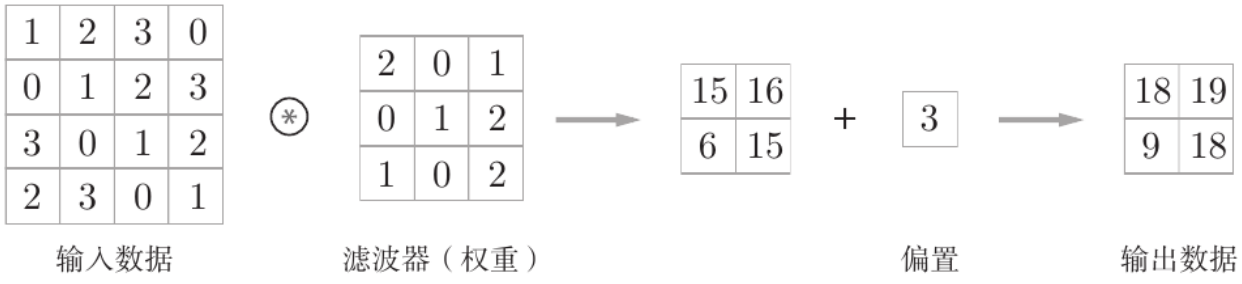
卷积运算

卷积运算相当于图像处理中的“滤波器运算”。

对于输入数据，卷积运算以一定间隔滑动滤波器的窗口并应用。将各个位置上滤波器的元素和输入的对应元素相乘，然后再求和（有时将这个计算称为乘积累加运算）。然后，将这个结果保存到输出的对应位置。将这个过程在所有位置都进行一遍，就可以得到卷积运算的输出。

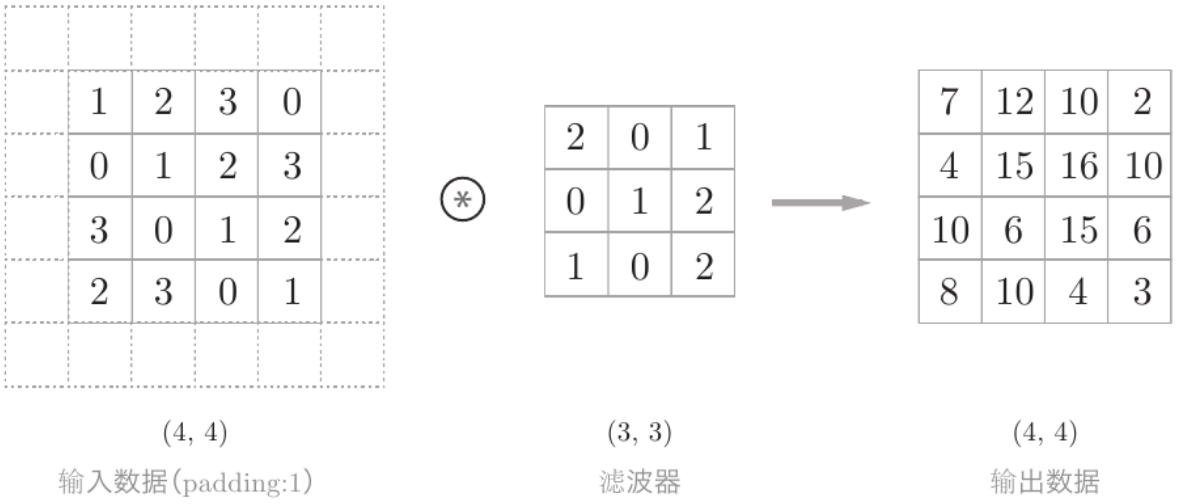


在全连接的神经网络中，除了权重参数，还存在偏置。CNN 中，滤波器的参数就对应之前的权重。并且，CNN 中也存在偏置。



填充

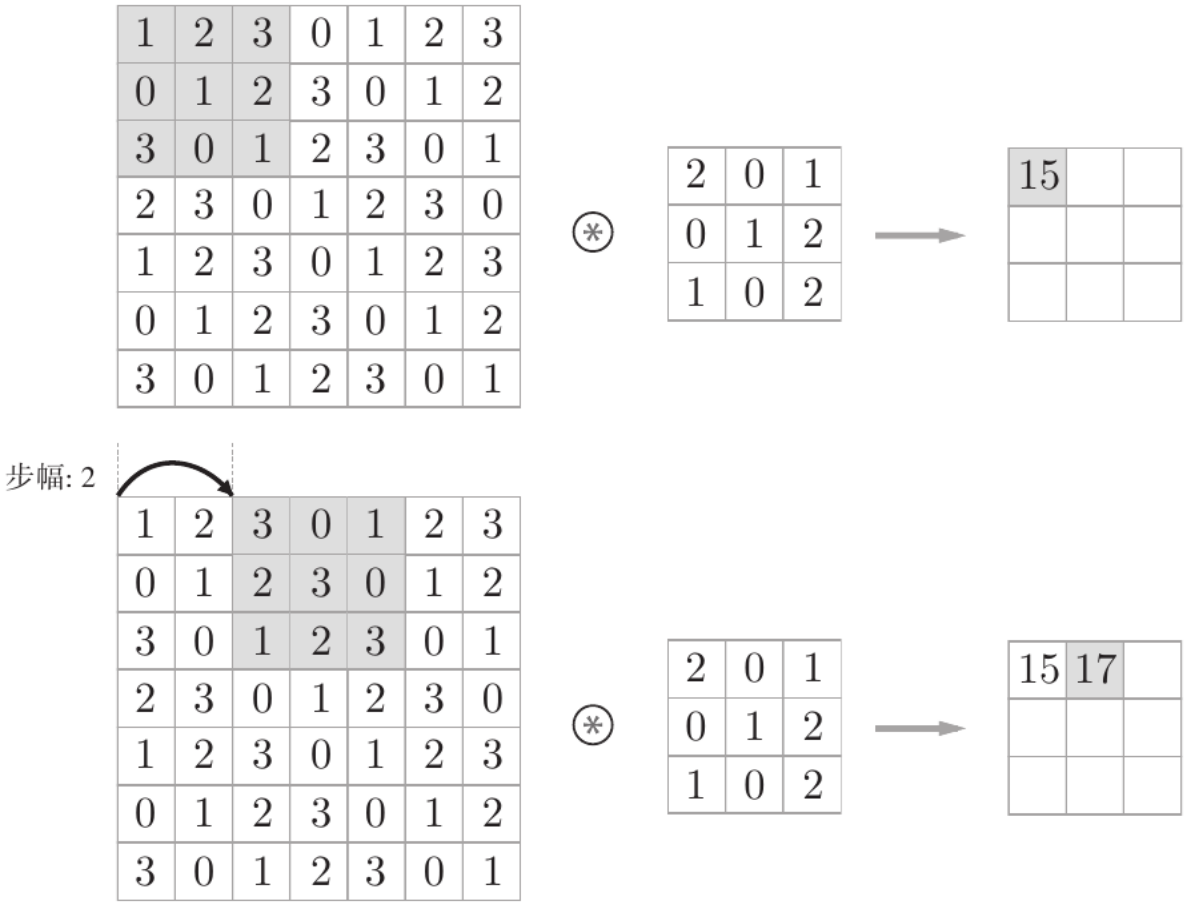
在进行卷积层的处理之前，有时要向输入数据的周围填入固定的数据（比如 0 等），这称为填充 （padding）。



使用填充主要是为了调整输出的大小。比如，对大小为 (4, 4) 的输入数据应用 (3, 3) 的滤波器时，输出大小变为 (2, 2)，相当于输出大小比输入大小缩小了 2 个元素。这在反复进行多次卷积运算的深度网络中会成为问题。为什么呢？因为如果每次进行卷积运算都会缩小空间，那么在某个时刻输出大小就有可能变为 1，导致无法再应用卷积运算。为了避免出现这样的情况，就要使用填充。在刚才的例子中，将填充的幅度设为 1，那么相对于输入大小 (4, 4)，输出大小也保持为原来的 (4, 4)。因此，卷积运算就可以在保持空间大小不变的情况下将数据传给下一层。

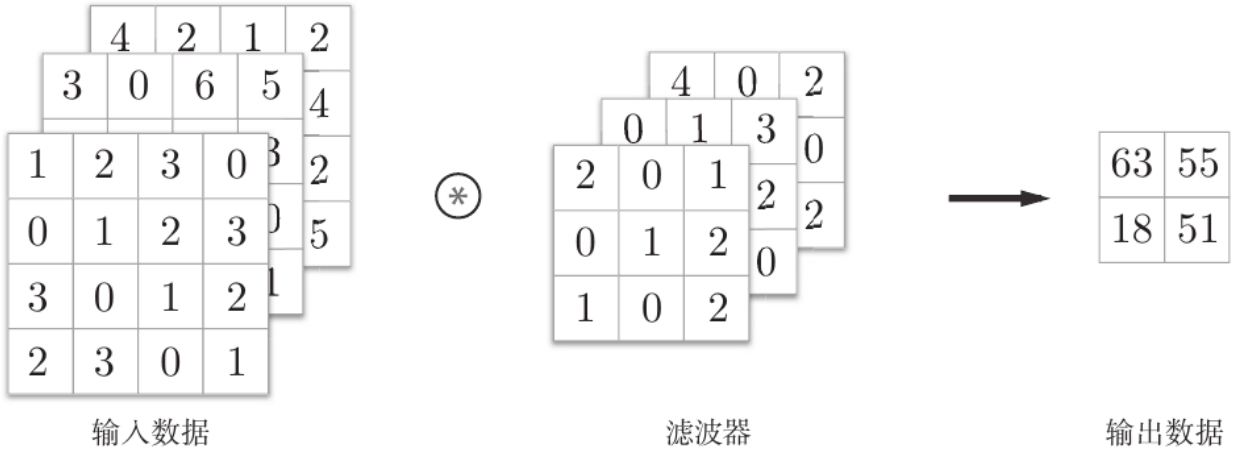
步幅

应用滤波器的位置间隔称为步幅 （stride）。之前的例子中步幅都是 1，如果将步幅设为 2，则如图所示



综上，增大步幅后，输出大小会变小。而增大填充后，输出大小会变大。

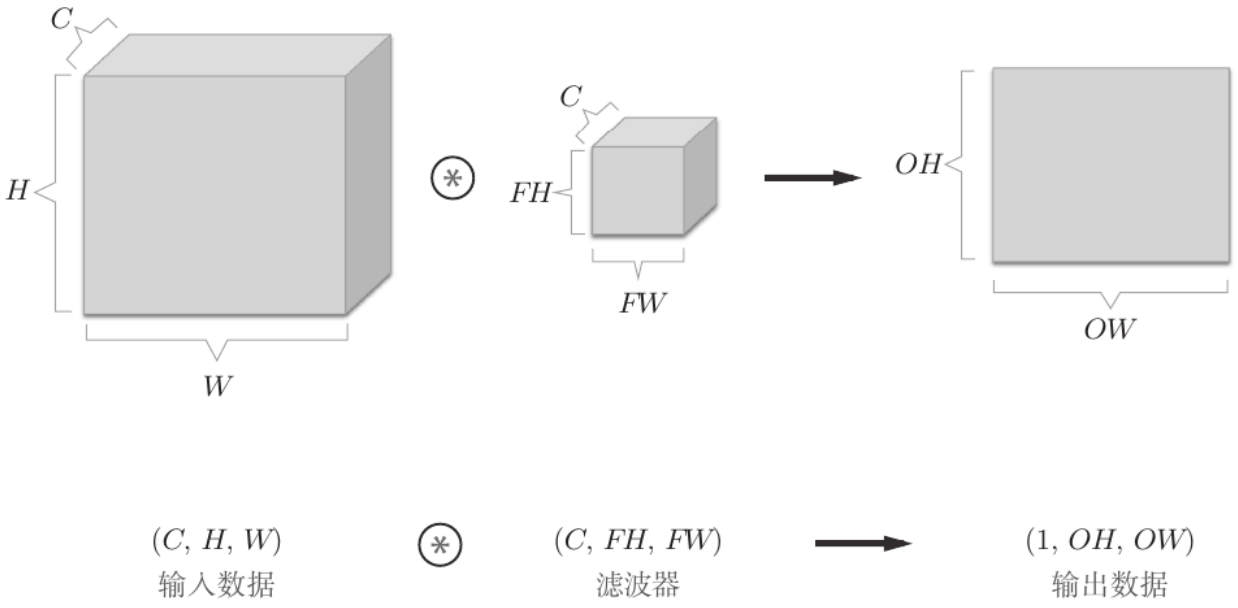
3维数据的卷积运算



需要注意的是，在 3 维数据的卷积运算中，输入数据和滤波器的通道数要设为相同的值。

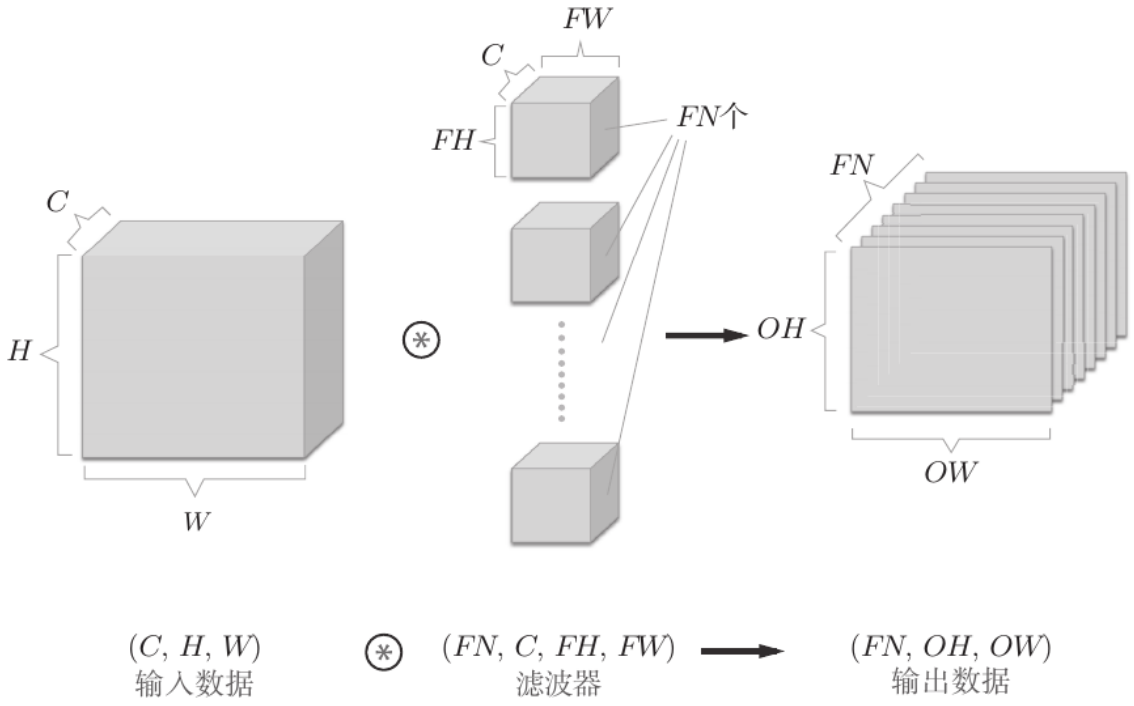
结合方块思考

通道数为C、高度为H、长度为W的数据的形状可以写成（C, H, W）。滤波器也一样，要按（channel,height,width）的顺序书写。比如，通道数为C、滤波器高度为FH（FilterHeight）、长度为FW （Filter Width）时，可以写成（C, FH, FW）。



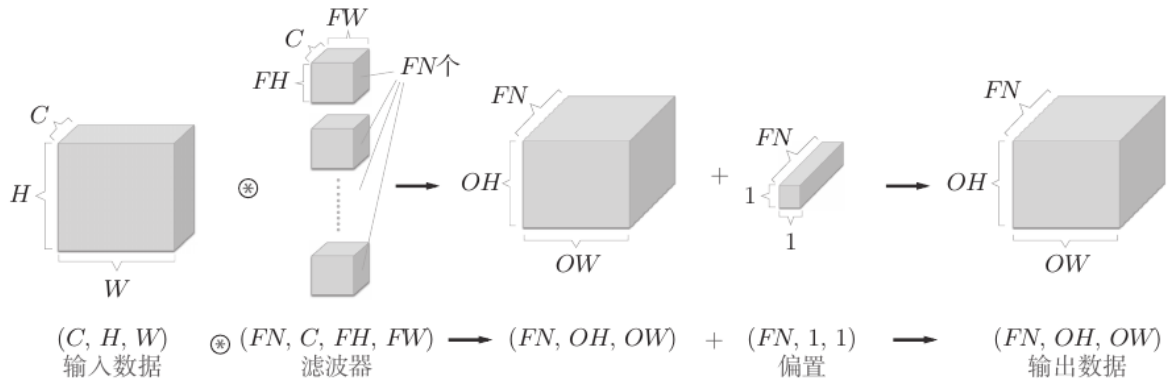
数据输出是1张特征图。所谓1张特征图，换句话说，就是通道数为1的特征图。

如果要在通道方向上也拥有多个卷积运算的输出，该怎么做呢？为此，就需要用到多个滤波器（权重）。

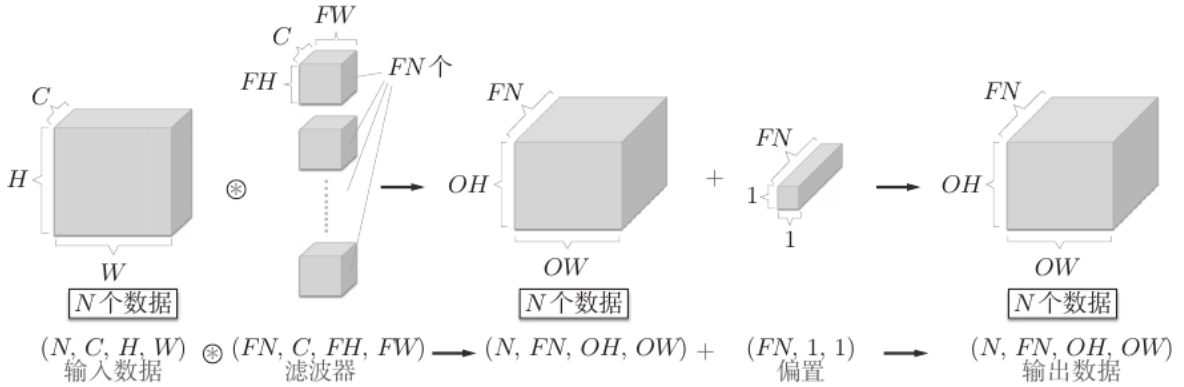


通过应用FN个滤波器，输出特征图也生成了FN个。如果将这FN个特征图汇集在一起，就得到了形状为(FN, OH, OW)的方块。将这个方块传给下一层，就是CNN的处理流。

卷积运算中（和全连接层一样）存在偏置。



批处理



这里，假设输入大小为 (H, W)，滤波器大小为 (FH, FW)，输出大小为(OH, OW)，填充为 P，步幅为 S。此时，输出大小可通过式进行计算。

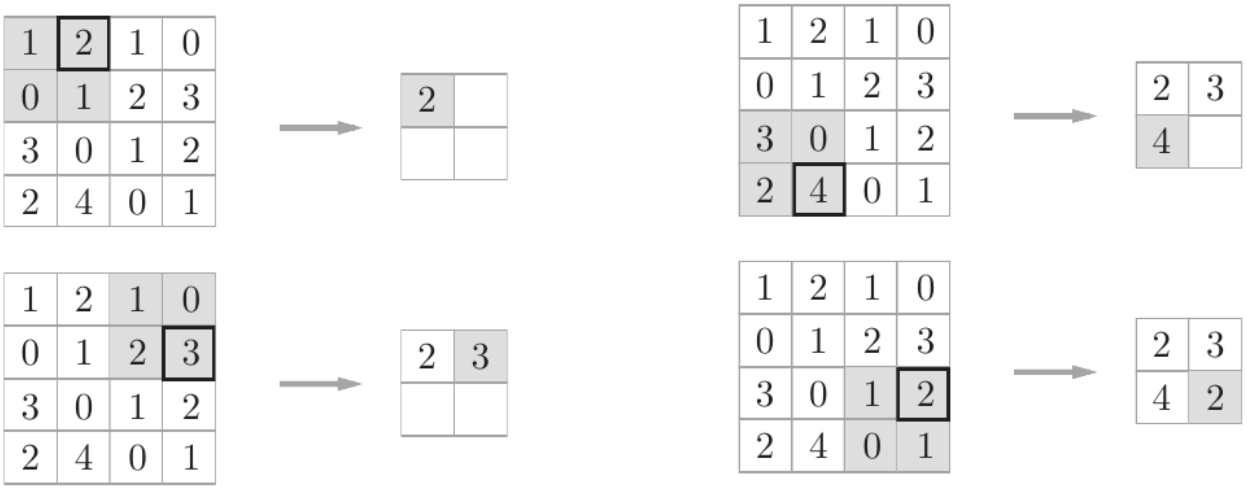
$$OH = \frac{H + 2P - FH}{S} + 1$$

$$OW = \frac{W + 2P - FH}{S} + 1$$

池化层

池化是缩小高、长方向上的空间的运算。

如图所示，从2 × 2 的区域中取出最大的元素。“Max池化”是获取最大值的运算，“2 × 2”表示目标区域的大小。



一般来说，池化的窗口大小会和步幅设定成相同的值。比如，3 × 3 的窗口的步幅会设为 3，4 × 4 的窗口的步幅会设为 4 等。

除了 Max 池化之外，还有 Average 池化等。相对于 Max 池化是从目标区域中取出最大值，Average 池化则是计算目标区域的平均值。在图像识别领域，主要使用 Max 池化。

池化层的特征：

- 没有要学习的参数
- 通道数不发生变化
- 对微小的位置变化具有鲁棒性（健壮）

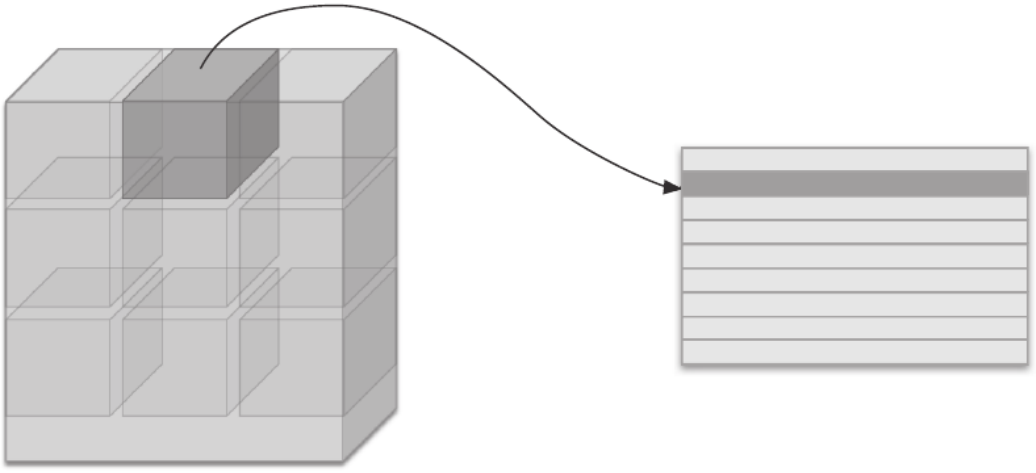
卷积层和池化层的实现

CNN 中处理的是 4 维数据，因此卷积运算的实现看上去会很复杂，但是通过使用下面要介绍的 im2col 这个技巧，问题就会变得很简单。

基于 im2col 的展开

如果老老实实地实现卷积运算，估计要重复好几层的 for 语句。这样的实现有点麻烦，而且， NumPy 中存在使用 for 语句后处理变慢的缺点（NumPy中，访问元素时最好不要用 for 语句）。

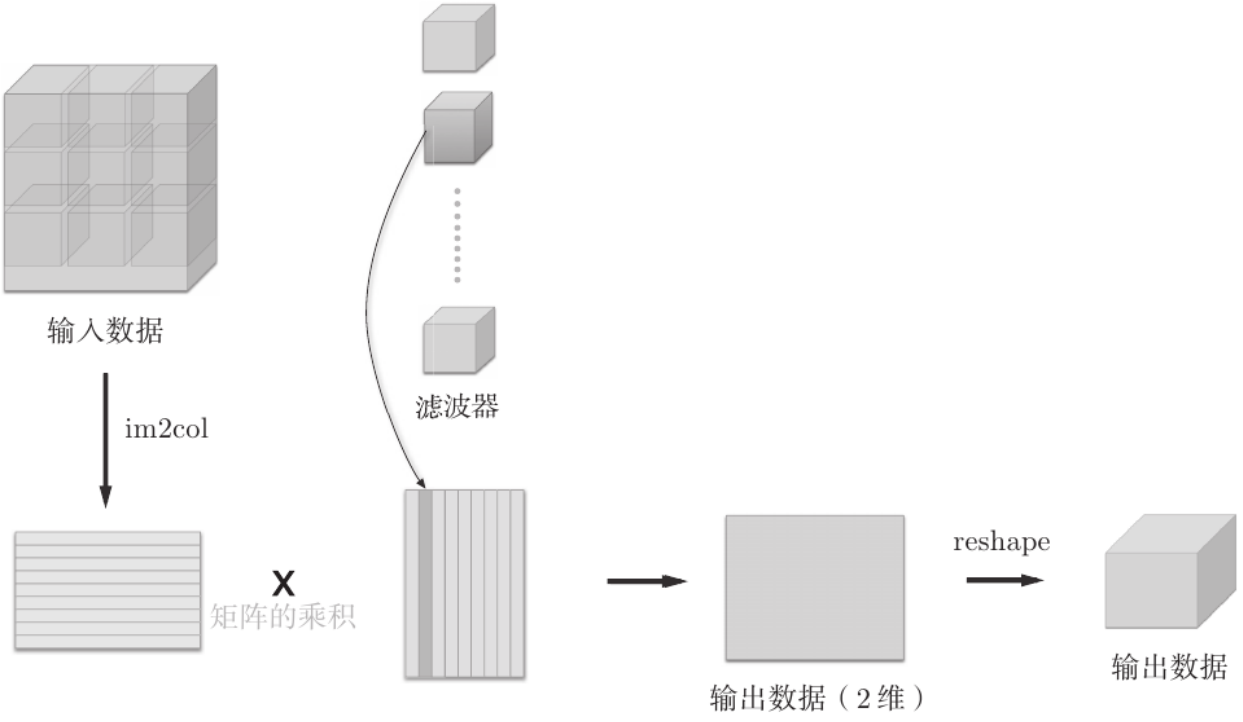
im2col 是一个函数，将输入数据展开以适合滤波器（权重）如图 7-17 所示，对 3 维的输入数据应用 im2col 后，数据转换为 2 维矩阵（正确地讲，是把包含批数量的 4 维数据转换成了 2 维数据）。



为了便于观察，将步幅设置得很大，以使滤波器的应用区域不重叠。而在实际的卷积运算中，滤波器的应用区域几乎都是重叠的。在滤波器的应用区域重叠的情况下，使用 im2col 展开后，展开后的元素个数会多于原方块的元素个数。因此，使用 im2col 的实现存在比普通的实现消耗更多内存的缺点。

im2col 这个名称是“image to column”的缩写，翻译过来就是“从图像到矩阵”的意思。Caffe、Chainer 等深度学习框架中有名为m2col 的函数，并且在卷积层的实现中，都使用了 im2col。

使用 im2col 展开输入数据后，之后就只需将卷积层的滤波器（权重）纵向展开为 1 列，并计算 2 个矩阵的乘积即可。这和全连接层的Affine 层进行的处理基本相同。



卷积层的实现

本书提供了 im2col 函数，并将这个 im2col 函数作为黑盒（不关心内部实现）使用。

im2col 这一便捷函数具有以下接口。

im2col (input_data, filter_h, filter_w, stride=1, pad=0)

- input_data —由（ 数据量，通道，高，长 ）的4 维数组构成的输入数据
- filter_h —滤波器的高
- filter_w —滤波器的长
- stride —步幅
- pad —填充

im2col 会考虑滤波器大小、步幅、填充，将输入数据展开为 2 维数组。

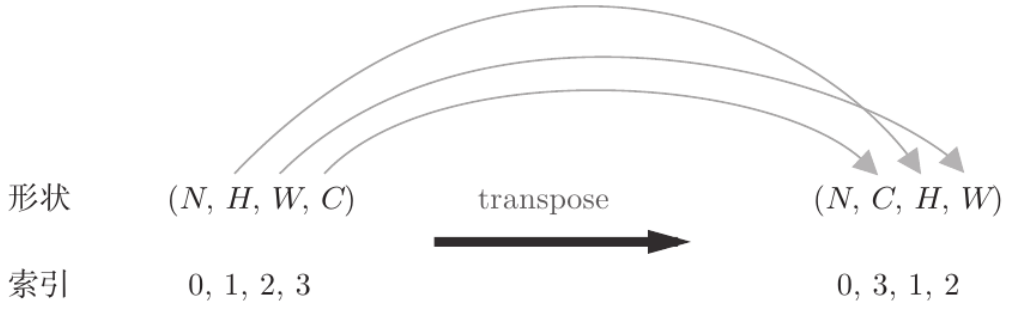
```
import sys, os
sys.path.append(os.pardir)
from common.util import im2col

x1 = np.random.rand(1, 3, 7, 7)
col1 = im2col(x1, 5, 5, stride=1, pad=0)
print(col1.shape) # (9, 75)

x2 = np.random.rand(10, 3, 7, 7) # 10 个数据
col2 = im2col(x2, 5, 5, stride=1, pad=0)
print(col2.shape) # (90, 75)
```

```
class Convolution:
    def __init__(self, W, b, stride=1, pad=0):
        self.W = W
        self.b = b
        self.stride = stride
        self.pad = pad
    def forward(self, x):
        FN, C, FH, FW = self.W.shape
        N, C, H, W = x.shape
        out_h = int(1 + (H + 2*self.pad - FH) / self.stride)
        out_w = int(1 + (W + 2*self.pad - FW) / self.stride)
        col = im2col(x, FH, FW, self.stride, self.pad)
```

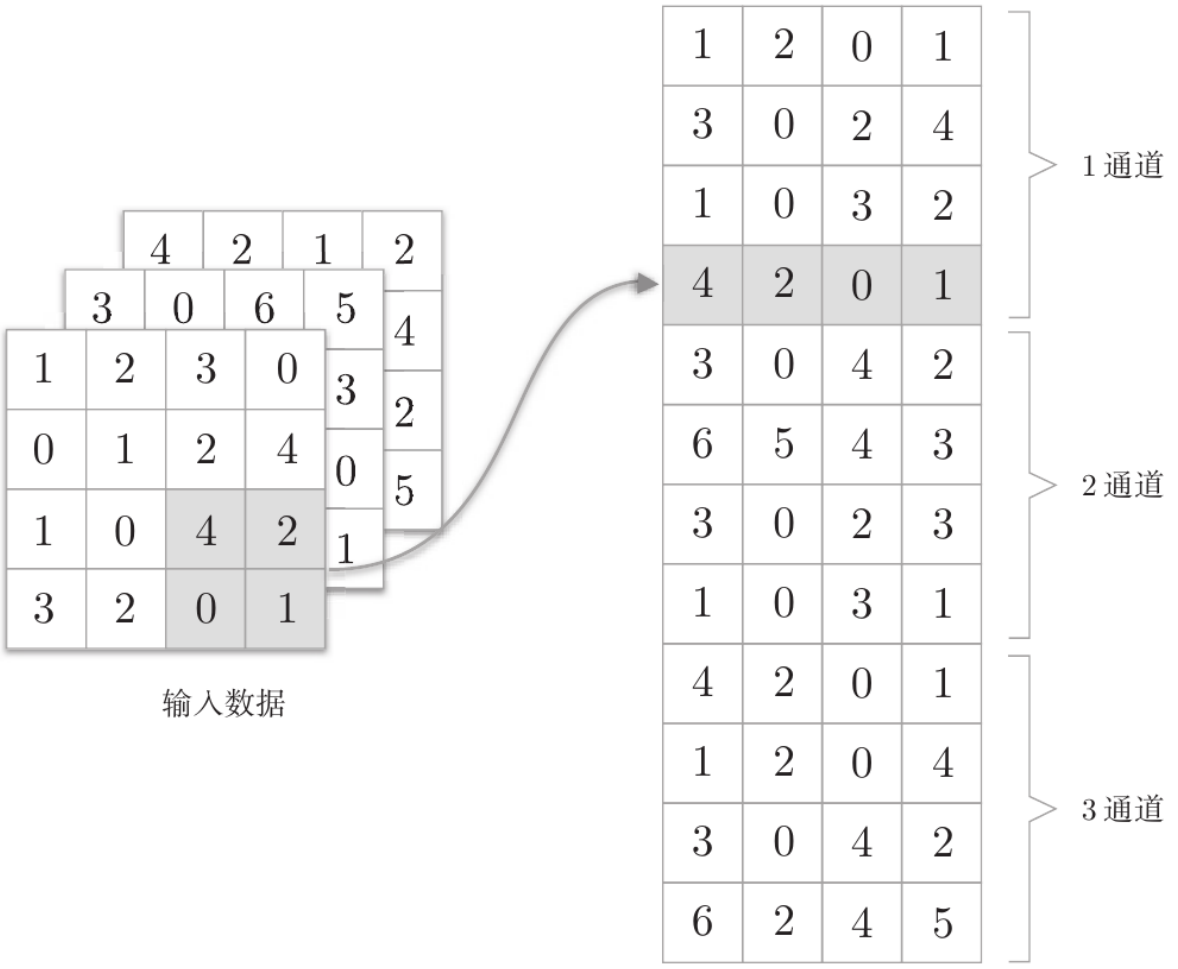
```
13          col_W = self.W.reshape(FN, -1).T # 滤波器的展开
14          out = np.dot(col, col_W) + self.b
15          out = out.reshape(N, out_h, out_w, -1).transpose(0, 3,
    1, 2)
16          return out
```

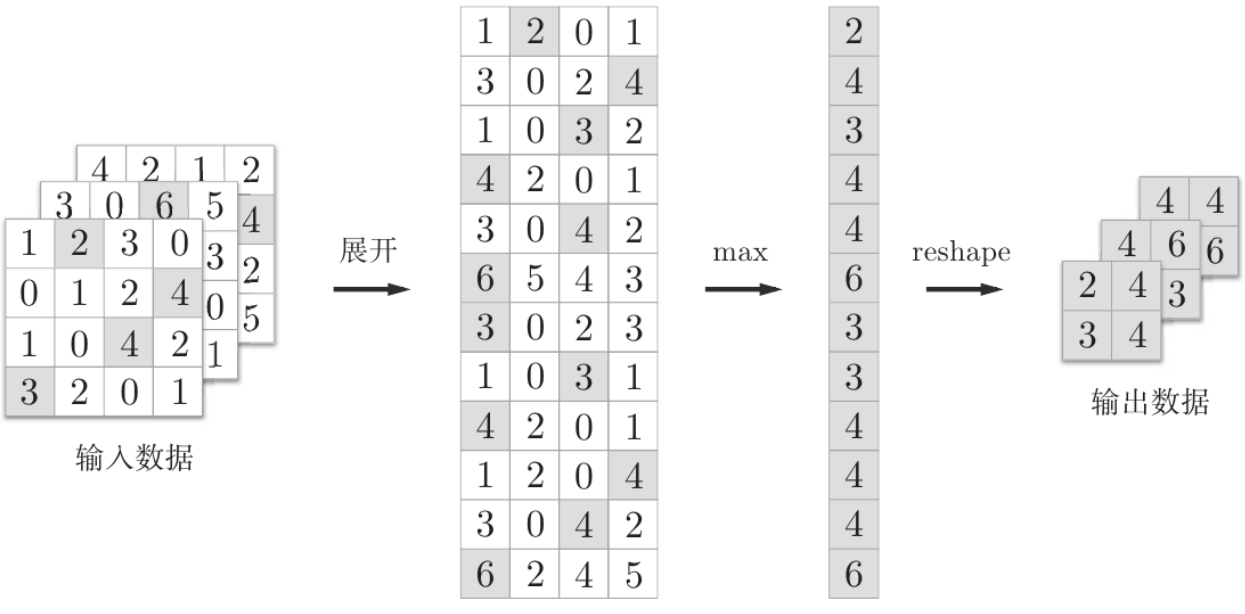
卷积层的初始化方法将滤波器（权重）、偏置、步幅、填充作为参数接收。滤波器是(FN, C, FH, FW)的4维形状。另外，FN、C、FH、FW分别是FilterNumber（滤波器数量）、Channel、Filter Height、Filter Width的缩写。

展开滤波器的部分（代码段中的粗体字）如图7-19所示，将各个滤波器的方块纵向展开为1列。这里通过reshape(FN,-1)将参数指定为-1，这是reshape的一个便利的功能。通过在reshape时指定为-1，reshape函数会自动计算-1维度上的元素个数，以使多维数组的元素个数前后一致。比如，(10, 3, 5, 5)形状的数组的元素个数共有750个，指reshape(10,-1)后，就会转换成(10, 75)形状的数组。



池化层的实现

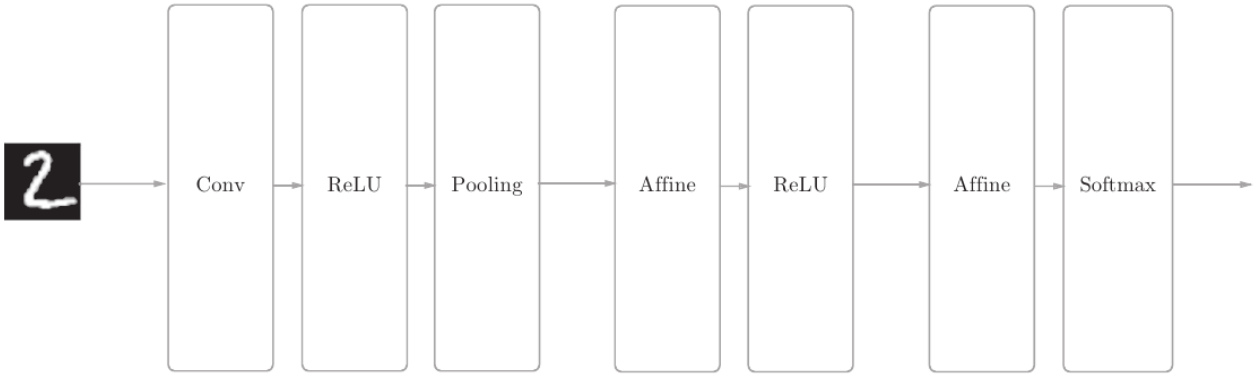





```python
class Pooling:
    def __init__(self, pool_h, pool_w, stride=1, pad=0):
        self.pool_h = pool_h
        self.pool_w = pool_w
        self.stride = stride
```

```
        self.pad = pad
    def forward(self, x):
        N, C, H, W = x.shape
        out_h = int(1 + (H - self.pool_h) / self.stride)
        out_w = int(1 + (W - self.pool_w) / self.stride)
        # 展开 (1)
        col = im2col(x, self.pool_h, self.pool_w, self.stride,
self.pad)
        col = col.reshape(-1, self.pool_h*self.pool_w)
        # 最大值 (2)
        out = np.max(col, axis=1)
        # 转换 (3)
        out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1,
2)
        return out
```

CNN的实现



网络的构成是“Convolution - ReLU - Pooling -Affine -ReLU - Affine - Softmax”，我们将它实现为名为 SimpleConvNet 的类。

SimpleConvNet 的初始化的实现稍长，我们分成 3 部分来说明，首先是初始化的最开始部分。

```
class SimpleConvNet:
    def __init__(self, input_dim=(1, 28, 28),
            conv_param={'filter_num':30, 'filter_size':5,
                        'pad':0, 'stride':1},
            hidden_size=100, output_size=10,
weight_init_std=0.01):
        filter_num = conv_param['filter_num']
        filter_size = conv_param['filter_size']
        filter_pad = conv_param['pad']
```

```
        filter_stride = conv_param['stride']
        input_size = input_dim[1]
        conv_output_size = (input_size - filter_size +
2*filter_pad) / \
            filter_stride + 1
        pool_output_size = int(filter_num * (conv_output_size/2)
*
                    (conv_output_size/2))
```

接下来是权重参数的初始化部分。

```
self.params = {}
self.params['W1'] = weight_init_std * \
    np.random.randn(filter_num, input_dim[0],
    filter_size, filter_size)
self.params['b1'] = np.zeros(filter_num)
self.params['W2'] = weight_init_std * \
    np.random.randn(pool_output_size,
    hidden_size)
self.params['b2'] = np.zeros(hidden_size)
self.params['W3'] = weight_init_std * \
    np.random.randn(hidden_size, output_size)
self.params['b3'] = np.zeros(output_size)
```

最后，生成必要的层。

```
self.layers = OrderedDict()
self.layers['Conv1'] = Convolution(self.params['W1'],
                                   self.params['b1'],
                                   conv_param['stride'],
                                   conv_param['pad'])
self.layers['Relu1'] = Relu()
self.layers['Pool1'] = Pooling(pool_h=2, pool_w=2, stride=2)
self.layers['Affine1'] = Affine(self.params['W2'],
                                self.params['b2'])
self.layers['Relu2'] = Relu()
self.layers['Affine2'] = Affine(self.params['W3'],
                                self.params['b3'])
self.last_layer = softmaxwithloss()
```

以上就是 SimpleConvNet 的初始化中进行的处理。像这样初始化后，进行推理的 predict 方法和求损失函数值的 loss 方法就可以像下面这样实现。

```
def predict(self, x):
    for layer in self.layers.values():
        x = layer.forward(x)
    return x
def loss(self, x, t):
    y = self.predict(x)
    return self.lastLayer.forward(y, t)
```

接下来是基于误差反向传播法求梯度的代码实现。

```
def gradient(self, x, t):
    # forward
    self.loss(x, t)
    # backward
    dout = 1
    dout = self.lastLayer.backward(dout)
    layers = list(self.layers.values())
    layers.reverse()
    for layer in layers:
        dout = layer.backward(dout)
    # 设定
    grads = {}
    grads['W1'] = self.layers['Conv1'].dW
    grads['b1'] = self.layers['Conv1'].db
    grads['W2'] = self.layers['Affine1'].dW
    grads['b2'] = self.layers['Affine1'].db
    grads['W3'] = self.layers['Affine2'].dW
    grads['b3'] = self.layers['Affine2'].db
    return grads
```

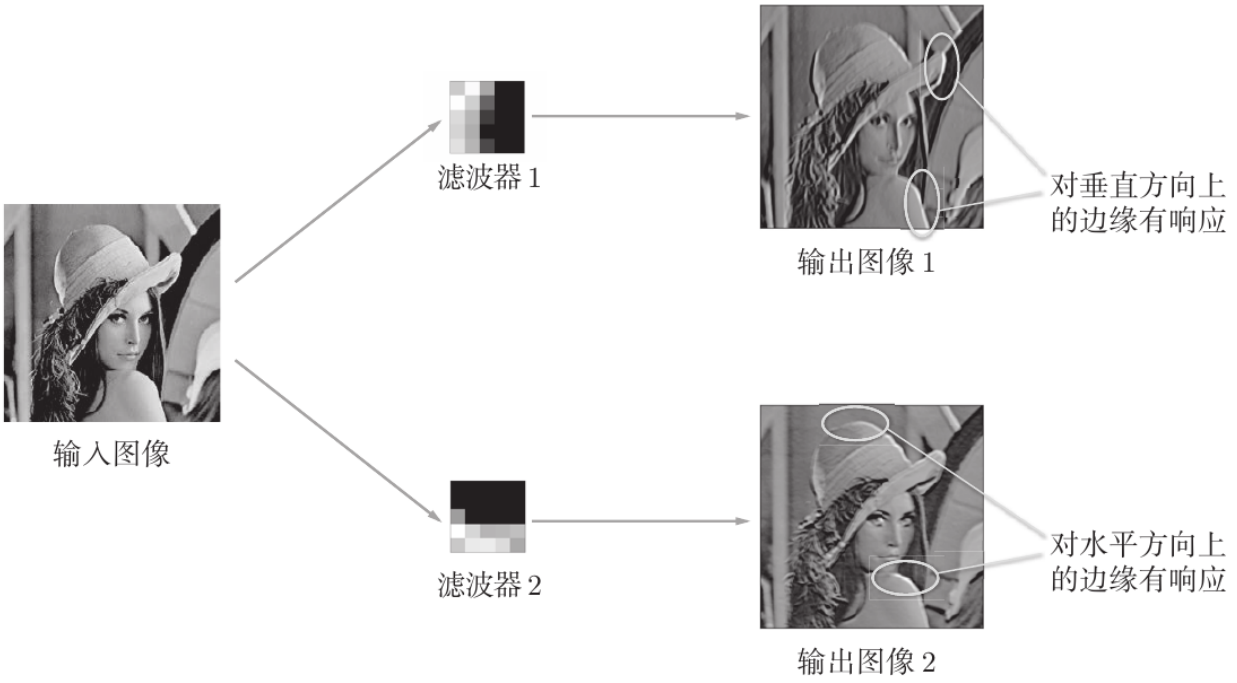
CNN 的可视化

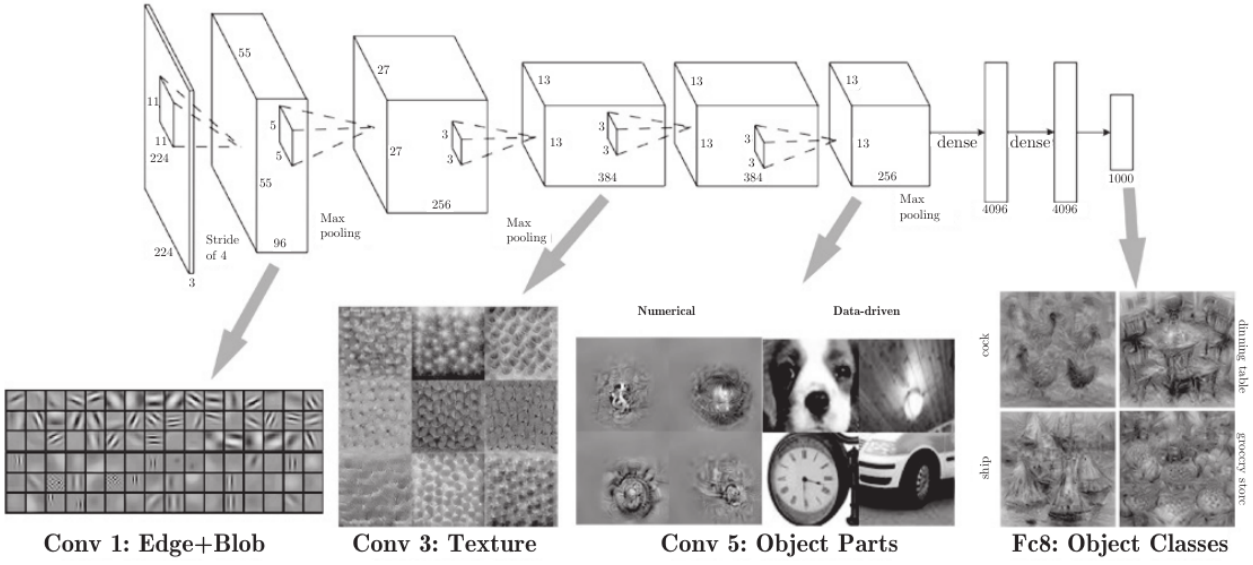
CNN 中用到的卷积层在“观察”什么呢？本节将通过卷积层的可视化，探索 CNN 中到底进行了什么处理。

卷积层的滤波器会提取边缘或斑块等原始信息。而刚才实现的 CNN 会将这些原始信息传递给后面的层。

上面的结果是针对第 1 层的卷积层得出的。第 1 层的卷积层中提取了边缘或斑块等“低级”信息，那么在堆叠了多层的 CNN 中，各层中又会提取什么样的信息呢？

如果堆叠了多层卷积层，则随着层次加深，提取的信息也愈加复杂、抽象，这是深度学习中很有意思的一个地方。最开始的层对简单的边缘有响应，接下来的层对纹理有响应，再后面的层对更加复杂的物体部件有响应。也就是说，随着层次加深，神经元从简单的形状向“高级”信息变化。换句话说，就像我们理解东西的“含义”一样，响应的对象在逐渐变化。



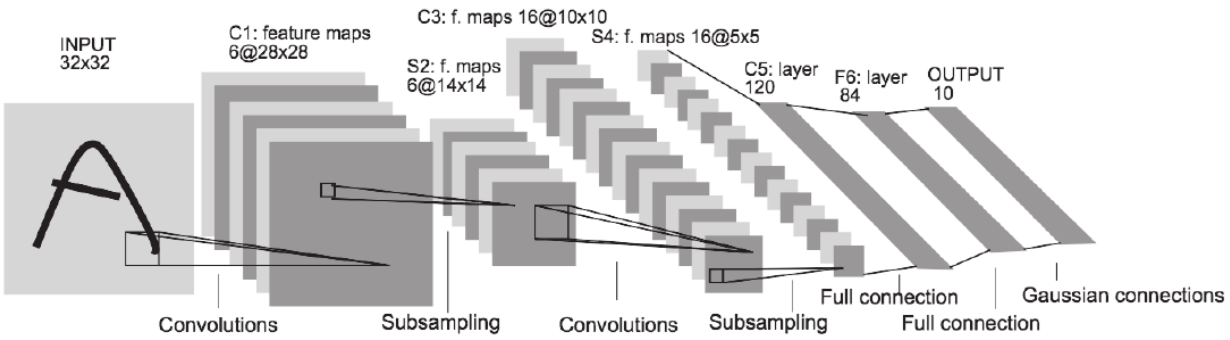


具有代表性的 CNN

这里，我们介绍其中特别重要的两个网络，一个是在 1998 年首次被提出的 CNN 元祖 LeNet，另一个是在深度学习受到关注的 2012 年被提出的 AlexNet。

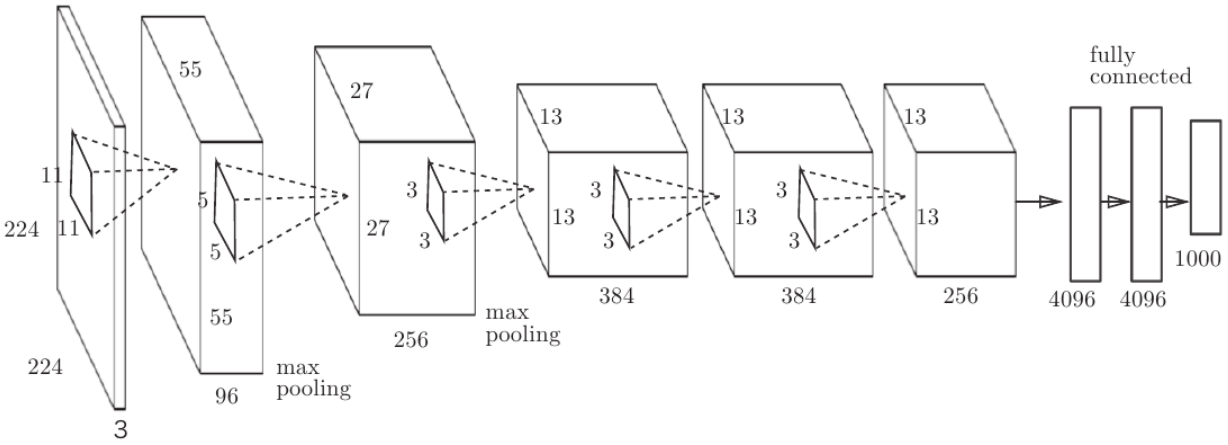
LeNet

LeNet 在 1998 年被提出，是进行手写数字识别的网络。如图 7-27 所示，它有连续的卷积层和池化层（正确地讲，是只“抽选元素”的子采样层），最后经全连接层输出结果。



和“现在的 CNN”相比，LeNet 有几个不同点。第一个不同点在于激活函数。LeNet 中使用 sigmoid 函数，而现在的 CNN 中主要使用 ReLU 函数。此外，原始的 LeNet 中使用子采样（subsampling)缩小中间数据的大小，而现在的 CNN 中 Max 池化是主流。

AlexNet



AlexNet 叠有多个卷积层和池化层，最后经由全连接层输出结果。虽然结构上 AlexNet 和 LeNet 没有大的不同，但有以下几点差异。

- 激活函数使用 ReLU。
- 使用进行局部正规化的 LRN （Local Response Normalization）层。
- 使用 Dropout 。

深度学习

深度学习是加深了层的深度神经网络。基于之前介绍的网络，只需通过叠加层，就可以创建深度网络。

加深层的动机

首先，从以 ILSVRC 为代表的大规模图像识别的比赛结果中可以看出加深层的重要性（详细内容请参考下一节）。这种比赛的结果显示，最近前几名的方法多是基于深度学习的，并且有逐渐加深网络的层的趋势。也就是说，可以看到层越深，识别性能也越高。

下面我们说一下加深层的好处。其中一个好处就是可以减少网络的参数数量。说得详细一点，就是与没有加深层的网络相比，加深了层的网络可以用更少的参数达到同等水平（或者更强）的表现力。

叠加小型滤波器来加深网络的好处是可以减少参数的数量，扩大感受野（receptive field，给神经元施加变化的某个局部空间区域）。并且，通过叠加层，将 ReLU 等激活函数夹在卷积层的中间，进一步提高了网络的表现力。这是因为向网络添加了基于激活函数的“非线性”表现力，通过非线性函数的叠加，可以表现更加复杂的东西。

加深层的另一个好处就是使学习更加高效。与没有加深层的网络相比，通过加深层，可以减少学习数据，从而高效地进行学习。也就是说，通过加深层，可以将各层要学习的问题分解成容易解决的简单问题，从而可以进行高效的学习。

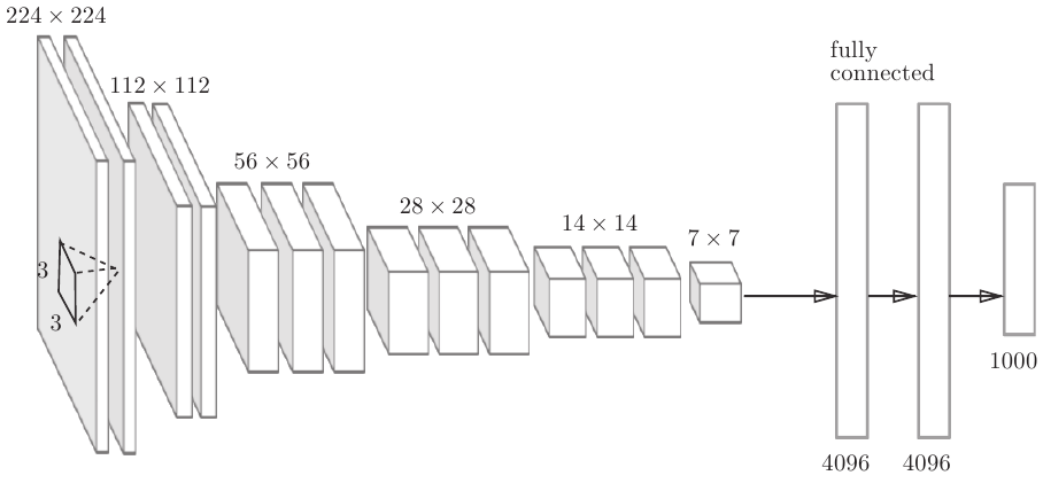
ImageNet

ImageNet是拥有超过 100 万张图像的数据集。这些年深度学习取得了不斐的成绩，其中VGG、GoogLeNet、ResNet已广为人知，在与深度学习有关的各种场合都会遇到这些网络。下面我们就来简单地介绍一下这 3 个有名的网络。

VGG

VGG 是由卷积层和池化层构成的基础的 CNN。它的特点在于将有权重的层（卷积层或者全连接层）叠加至 16 层（或者 19 层），具备了深度（根据层的深度，有时也称为“VGG16”或“VGG19”）。

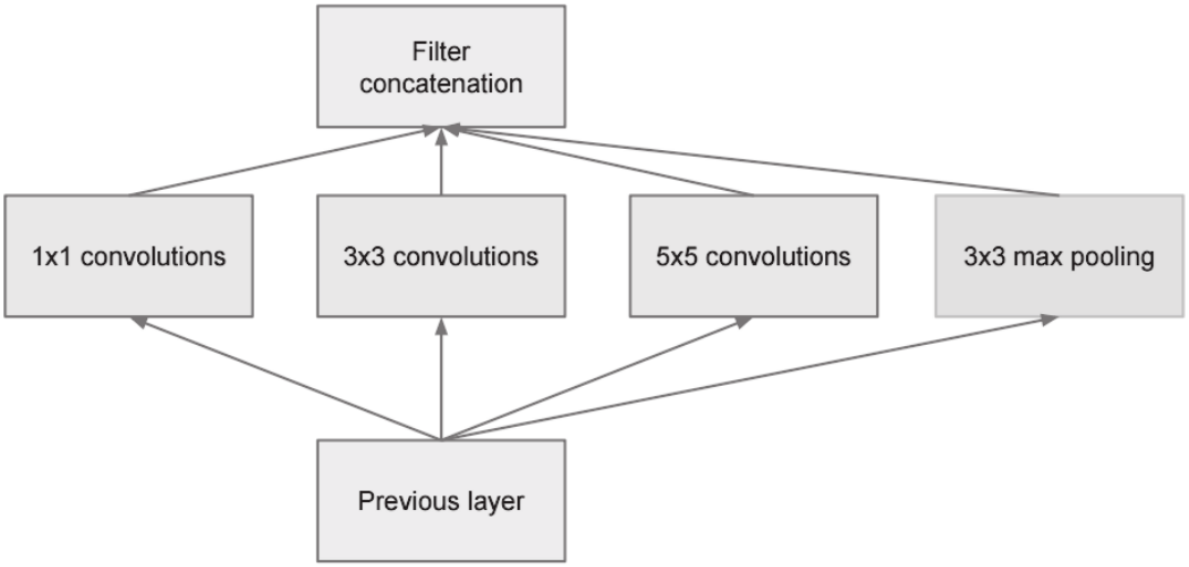
VGG 中需要注意的地方是，基于 3×3 的小型滤波器的卷积层的运算是连续进行的。如图所示，重复进行“卷积层重叠 2 次到 4 次，再通过池化层将大小减半”的处理，最后经由全连接层输出结果。



GoogLeNet

GoogLeNet 的特征是，网络不仅在向上有深度，在横向上也有深度（广度）。

GoogLeNet 在横向上有“宽度”，这称为“Inception 结构”。Inception 结构使用了多个大小不同的滤波器（和池化），最后再合并它们的结果。GoogLeNet 的特征就是将这个 Inception 结构用作一个构件（构成元素）。此外，在 GoogLeNet 中，很多地方都使用了大小为1 × 1 的滤波器的卷积层。这个 1 × 1 的卷积运算通过在通道方向上减小大小，有助于减少参数和实现高速化处理。

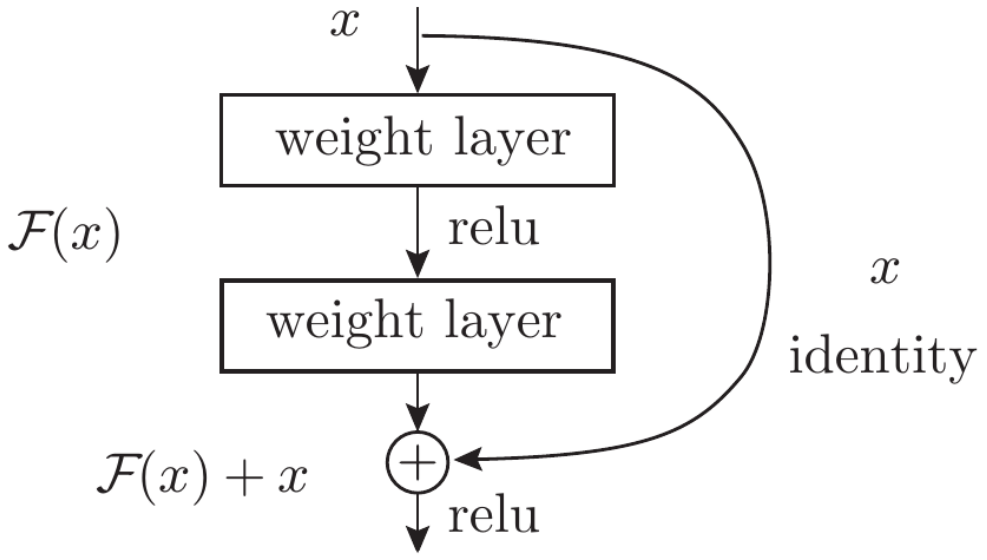


ResNet

它的特征在于具有比以前的网络更深的结构。

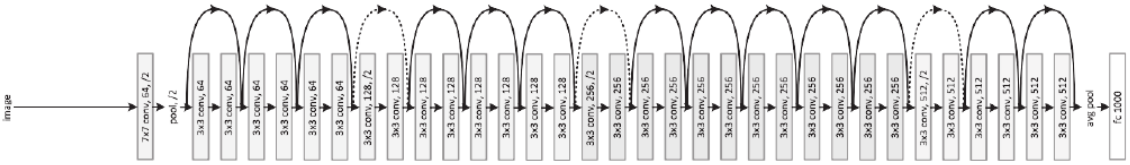
我们已经知道加深层对于提升性能很重要。但是，在深度学习中，过度加深层的话，很多情况下学习将不能顺利进行，导致最终性能不佳。ResNet 中，为了解决这类问题，导入了“快捷结构”（也称为“捷径”或“小路”）。导入这个快捷结构后，就可以随着层的加深而不断提高性能了（当然，层的加深也是有限度的）。

快捷结构横跨（跳过）了输入数据的卷积层，将输入 x 合计到输出。



因为快捷结构只是原封不动地传递输入数据，所以反向传播时会将来自上游的梯度原封不动地传向下游。这里的重点是不对来自上游的梯度进行任何处理，将其原封不动地传向下游。因此，基于快捷结构，不用担心梯度会变小（或变大），能够向前一层传递“有意义的梯度”。通过这个快捷结构，之前因为加深层而导致的梯度变小的梯度消失问题就有望得到缓解。

ResNet 以前面介绍过的 VGG 网络为基础，引入快捷结构以加深层。



实践中经常会灵活应用使用 ImageNet 这个巨大的数据集学习到的权重数据，这称为迁移学习，将学习完的权重（的一部分）复制到其他神经网络，进行再学习（fine tuning）。比如，准备一个和 VGG 相同结构的网络，把学习完的权重作为初始值，以新数据集为对象，进行再学习。迁移学习在手头数据集较少时非常有效。