

深度学习入门

基于Python的理论与实现

Stephen CUI

March 14, 2023

Contents

1 感知机	1
1.1 感知机是什么	1
1.2 简单逻辑电路	1
1.2.1 与门	1
1.2.2 与非门和或门	1
1.3 感知机的实现	2
1.3.1 简单的实现	2
1.3.2 导入权重和偏置	2
1.4 感知机的局限性	2
1.4.1 异或门	2
1.4.2 线性和非线性	2
1.5 多层感知机	2
1.5.1 已有门电路的组合	3
1.5.2 异或门的实现	3
1.6 从与非门到计算机	3
2 神经网络	4
2.1 从感知机到神经网络	4
2.1.1 神经网络的例子	4
2.1.2 复习感知机	4
2.1.3 激活函数登场	4
2.2 激活函数	5
2.2.1 sigmoid函数	5
2.2.2 sigmoid函数和阶跃函数的比较	5
2.2.3 非线性函数	6
2.2.4 ReLU函数	6
2.3 多维数组的运算	6
2.4 3层神经网络的实现	6
2.5 输出层的设计	8
2.5.1 恒等函数和softmax函数	8
2.5.2 实现softmax函数时的注意事项	8

2.5.3 softmax函数的特征	8
2.5.4 输出层的神经元数量	8
2.6 手写数字识别	8
2.6.1 批处理	9
2.7 小结	9
3 神经网络的学习	10
3.1 从数据中学习	10
3.1.1 数据驱动	10
3.1.2 训练数据和测试数据	10
3.2 损失函数	11
3.2.1 均方误差	11
3.2.2 交叉熵误差	11
3.2.3 mini-batch学习	11
3.2.4 mini-batch版交叉熵误差的实现	12
3.2.5 为何要设定损失函数	12
3.3 数值微分	12
3.4 梯度	12
3.4.1 梯度法	13
3.4.2 神经网络的梯度	14
3.5 学习算法的实现	14
3.5.1 基于测试数据的评价	14
4 误差反向传播法	16
4.1 计算图	16
4.1.1 局部计算	16
4.2 链式法则	17
4.2.1 计算图的反向传播	17
4.2.2 什么是链式法则	17
4.2.3 链式法则和计算图	17
4.3 反向传播	17
4.3.1 加法节点的反向传播	18
4.3.2 乘法节点的反向传播	18
4.4 简单层的实现	19
4.4.1 乘法层的实现	19
4.5 激活函数层的实现	19
4.5.1 ReLU层	19
4.5.2 Sigmoid层	19
4.6 Affine/Softmax层的实现	20
4.6.1 Affine层	20
4.6.2 批版本的Affine层	21

4.6.3 Softmax-with-Loss 层	21
4.7 误差反向传播法的实现	22
4.7.1 误差反向传播法的梯度确认	22
4.7.2 使用误差反向传播法的学习	23
5 与学习相关的技巧	24
5.1 参数的更新	24
5.1.1 SGD	24
5.1.2 SGD的缺点	24
5.1.3 Momentum	24
5.1.4 AdaGrad	25
5.1.5 Adam	25
5.1.6 使用哪种更新方法呢	26
5.1.7 基于MNIST数据集的更新方法的比较	26
5.2 权重的初始值	26
5.2.1 可以将权重初始值设为0吗	26
5.2.2 隐藏层的激活值的分布	27
5.2.3 ReLU的权重初始值	28
5.2.4 基于MNIST数据集的权重初始值的比较	28
5.3 Batch Normalization	28
5.3.1 Batch Normalization 的算法	28
5.3.2 Batch Normalization的评估	30
5.4 正则化	31
5.4.1 过拟合	31
5.4.2 权值衰减	32
5.4.3 Dropout	32
5.5 超参数的验证	34
5.5.1 验证数据	34
5.5.2 超参数的最优化	34
5.5.3 超参数最优化的实现	35
5.6 小结	35
6 卷积神经网络	36
6.1 整体结构	36
6.2 卷积层	36
6.2.1 全连接层存在的问题	36
6.2.2 卷积运算	37
6.2.3 填充	39
6.2.4 步幅	39
6.2.5 3维数据的卷积运算	40
6.2.6 结合方块思考	42

6.2.7 批处理	43
6.3 池化层	43
6.4 卷积层和池化层的实现	44
6.4.1 4维数组	44
6.4.2 基于im2col的展开	44
6.4.3 卷积层的实现	45
6.4.4 池化层的实现	46
6.5 CNN的实现	46
6.6 CNN的可视化	46
6.6.1 第1层权重的可视化	46
6.6.2 基于分层结构的信息提取	46
6.7 具有代表性的CNN	46
6.7.1 LeNet	46
6.7.2 AlexNet	48
6.8 小结	48
7 深度学习	49
7.1 加深网络	49
7.1.1 向更深的网络出发	49
7.1.2 进一步提高识别精度	49
7.1.3 加深层的动机	50
7.2 深度学习的小历史	51
7.2.1 VGG	51
7.2.2 GoogLeNet	52
7.2.3 ResNet	52
7.3 深度学习的高速化	54
7.3.1 需要努力解决的问题	54
7.3.2 基于GPU的高速化	54
7.3.3 分布式学习	54
7.3.4 运算精度的位数缩减	54
7.4 深度学习的应用案例	54
7.4.1 物体检测	54
7.4.2 图像分割	54
7.4.3 图像标题生成	55
7.5 深度学习的未来	55
7.5.1 图像风格变换	55
7.5.2 图像的生成	55
7.5.3 自动驾驶	56
7.5.4 Deep Q-Network(强化学习)	56

A Softmax-with-Loss层的计算图	57
A.1 反向传播	57
A.1.1 Cross Entropy Error层	57
A.1.2 Softmax层	57

List of Figures

2.1 Examples of Neural Networks	5
2.2 Signal passing from input layer to layer 1	6
2.3 Layer 1 to Layer 2 Signaling	7
2.4 Signal passing from layer 2 to output layer	7
2.5 change of array shape	9
2.6 Variation of array shape in batch	9
3.1 Moving from human designed rules to machine learning from data	10
3.2 2D Gradient Schematic	12
3.3 The update process of the gradient method	14
4.1 Based on the calculation graph to solve the answer	16
4.2 Based on the calculation graph to solve the answer2	17
4.3 Backpropagation for Adder nodes	18
4.4 Backpropagation of multiply nodes	18
4.5 Example of backpropagation for buying apples	19
4.6 Computational graph of the Sigmoid layer	20
4.7 Computational graph of the Affine layer	20
4.8 Computational graph of the batch version of the Affine layer	21
4.9 Computational graph of the Softmax-with-Loss layer	21
4.10 Simple version of the calculation graph of the Softmax-with-Loss layer	22
5.1 Momentum-The ball rolls on an incline	25
5.2 Comparison of Optimization Methods	26
5.3 The distribution of the activation value of each layer when using a Gaussian distribution with a standard deviation of 1 as the initial weight value	27
5.4 Xavier initial value	28
5.5 Distribution of activation values of each layer when using Xavier initial value as weight initial value	29
5.6 Comparison of weight initial values based on MNIST dataset	29
5.7 An example of a neural network using Batch Normalization	30
5.8 Computational graph of Batch Normalization	30

5.9	Use batch norm to compare with no use	31
5.10	Comparison of recognition accuracy between training data and test data	31
5.11	Changes in recognition accuracy of training data and test data using weight decay	32
5.12	Concept map of Dropout	33
5.13	Comparison between dropout use or not	33
5.14	best 20 workouts	35
6.1	An example of a network based on a fully connected layer (Affine layer)	36
6.2	Examples of CNN-based networks	37
6.3	Example of convolution operation	37
6.4	Calculation order of convolution operation	38
6.5	The bias of the convolution operation	38
6.6	Filling processing of convolution operation	39
6.7	Example of convolution operation with stride 2	40
6.8	Example of convolution operation on 3D data	40
6.9	Computational order for convolution operations on 3D data	41
6.10	Thinking about convolution operations in combination with blocks	42
6.11	Example of convolution operation based on multiple filters	42
6.12	Processing flow of convolution operation	43
6.13	Processing flow of convolution operation (batch processing)	43
6.14	The processing order of Max pooling	44
6.15	Features of the pooling layer	44
6.16	function im2col	45
6.17	Details of filter processing for convolution operations	45
6.18	The implementation process of the pooling layer	46
6.19	Network composition of a simple CNN	46
6.20	CNN training results on MNIST	47
6.21	The network structure of LeNet	47
6.22	AlexNet	48
7.1	Deep CNN for Handwritten Digit Recognition	49
7.2	Examples of misidentified images	50
7.3	Data Augmentation example	50
7.4	5-5 convolution example	51
7.5	Example of a convolutional layer repeated twice 3-3	51
7.6	VGG	52
7.7	GoogLeNet	52
7.8	Inception structure of GoogLeNet	53
7.9	Components of ResNet	53
7.10	Processing flow of R-CNN	54
7.11	Neural Image Caption	55

A.1 Step 3	58
----------------------	----

Chapter 1

感知机

本章将介绍感知机（perceptron）这一算法。感知机作为神经网络（深度学习）的起源的算法。因此，学习感知机的构造是学习通向神经网络和深度学习的一种重要思想。

1.1 感知机是什么

感知机接收多个输入信号，输出一个信号。像电流流过导线，向前方输送电子一样，感知机的信号也会形成流，向前方输送信息。但是，和实际的电流不同的是，感知机的信号只有“流/不流”（1/0）两种取值。在本书中，0 对应“不传递信号”，1 对应“传递信号”。

假设一个接收两个输入信号的感知机的例子。 x_1 、 x_2 是输入信号， y 是输出信号， w_1 、 w_2 是权重（ w 是 weight 的首字母）。输入信号被送往神经元时，会被分别乘以固定的权重(w_1x_1, w_2x_2)，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。这也称为“神经元被激活”。这里将这个界限值称为阈值，用符号 θ 表示。

感知机的运行原理只有这些！把上述内容用数学式来表示：

$$y = \begin{cases} 0, & w_1x_1 + w_2x_2 \leq \theta \\ 1, & w_1x_1 + w_2x_2 > \theta \end{cases} \quad (1.1)$$

感知机的多个输入信号都有各自固有的权重，这些权重发挥着控制各个信号的重要性的作用。也就是说，权重越大，对应该权重的信号的重要性就越高。

1.2 简单逻辑电路

1.2.1 与门

与门（AND gate）是有两个输入和一个输出的门电路。与门仅在两个输入均为1时输出1，其他时候则输出0。

1.2.2 与非门和或门

与非门（NAND gate）就是颠倒了与门的输出，仅当 x_1 和 x_2 同时为1时输出0，其他时候则输出1。

或门是“只要有一个输入信号是1，输出就为1”的逻辑电路。

这里决定感知机参数的并不是计算机，而是我们人。我们看着真值表这种“训练数据”，人工考虑（想到）了参数的值。而机器学习的课题就是将这个决定参数值的工作交由计算机自动进行。学习是确定合适的参数的过程，而人要做的是思考感知机的构造（模型），并把训练数据交给计算机。

与门、与非门、或门的感知机构造是一样的。实际上，3个门电路只有参数的值（权重和阈值）不同。也就是说，相同构造的感知机，只需通过适当地调整参数的值，就可以不断变换为与门、与非门、或门。

1.3 感知机的实现

1.3.1 简单的实现

1.3.2 导入权重和偏置

首先把 Equation 1.1 的 θ 换成 $-b$ ，就可以用 Equation 1.2 来表示感知机的行为。

$$y = \begin{cases} 0, & b + w_1x_1 + w_2x_2 \leq 0 \\ 1, & b + w_1x_1 + w_2x_2 > 0 \end{cases} \quad (1.2)$$

这里， b 称为偏置， w_1 和 w_2 称为权重。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。请注意，偏置和权重 w_1 、 w_2 的作用是不一样的。具体地说， w_1 和 w_2 是控制输入信号的重要性的参数，而偏置是调整神经元被激活的容易程度（输出信号为 1 的程度）的参数。

1.4 感知机的局限性

1.4.1 异或门

异或门也被称为逻辑异或电路。仅当 x_1 或 x_2 中的一方为 1 时，才会输出 1（“异或”是拒绝其他的意思）。实际上，用前面介绍的感知机是无法实现这个异或门的。

可以考虑在二维平面上添加一个直线，但是没有一个直线能够满足将 $(0,0), (0,1), (1,0), (1,1)$ 实现异或门的切分。

1.4.2 线性和非线性

显然，我们可以使用抛物线这种非线性的曲线来实现异或门，这就是用非线性的方式来实现的。**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。**由曲线分割而成的空间称为非线性空间，由直线分割而成的空间称为线性空间。

1.5 多层感知机

感知机不能表示异或门让人深感遗憾，但也无需悲观。实际上，感知机的绝妙之处在于它可以“叠加层”，这样可以实现异或门。

1.5.1 已有门电路的组合

感知机的局限性，严格地讲，应该是“单层感知机无法表示异或门”或者“单层感知机无法分离非线性空间”。接下来，我们将看到通过组合感知机（叠加层）就可以实现异或门。

异或门可以通过与门、与非门、或门组成的两层感知机来实现，首先是第一层：使用非门和与非门来作为输入，然后第二层使用非门和与非门的输出作为与门的输入，即可实现异或门。

1.5.2 异或门的实现

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次感知机（multi-layered perceptron, MLP）。

1.6 从与非门到计算机

计算机是处理信息的机器。向计算机中输入一些信息后，它会按照某种既定的方法进行处理，然后输出结果。所谓“按照某种既定的方法进行处理”是指，计算机和感知机一样，也有输入和输出，会按照某个既定的规则进行计算。

Chapter 2

神经网络

关于感知机，既有好消息，也有坏消息。好消息是，即便对于复杂的函数，感知机也隐含着能够表示它的可能性。上一章已经介绍过，即便是计算机进行的复杂处理，感知机（理论上）也可以将其表示出来。坏消息是，设定权重的工作，即确定合适的、能符合预期的输入与输出的权重，现在还是由人工进行的。

2.1 从感知机到神经网络

2.1.1 神经网络的例子

用图来表示神经网络的话，如Figure 2.1所示。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。另外，本书中把输入层到输出层依次称为第0层、第1层、第2层。

2.1.2 复习感知机

引入新函数 $h(x)$ ，将Equation 1.2改写成下面的方程：

$$y = h(b + w_1x_1 + w_2x_2) \quad (2.1)$$

$$h(x) = \begin{cases} 0 & , x \leq 0 \\ 1 & , x > 0 \end{cases} \quad (2.2)$$

2.1.3 激活函数登场

Equation 2.2中的 $h(x)$ 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）。

本书在使用“感知机”一词时，没有严格统一它所指的算法。一般而言，“朴素感知机”是指单层网络，指的是激活函数使用了阶跃函数A的模型。“多层感知机”是指神经网络，即使用sigmoid函数等平滑的激活函数的多层网络。

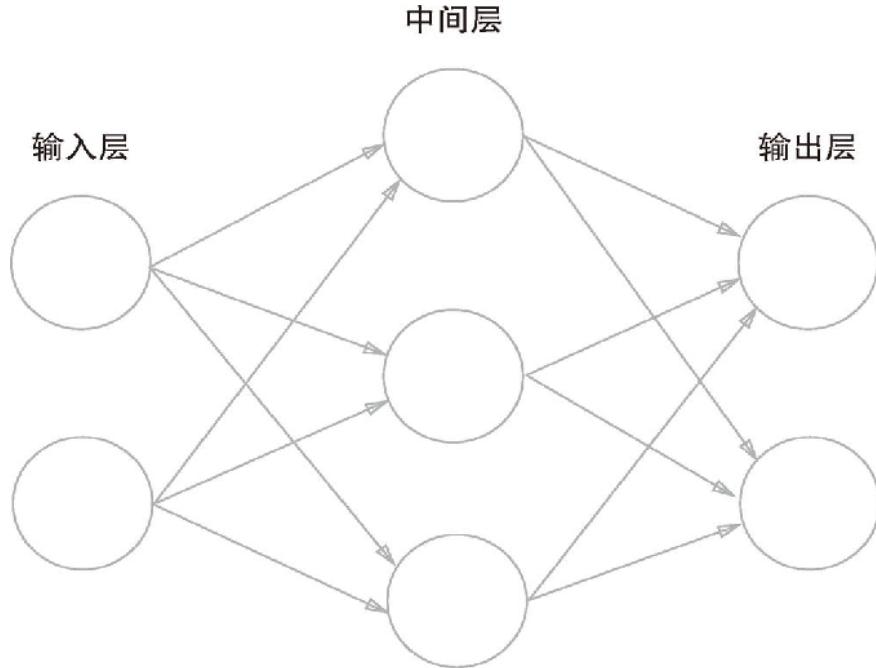


Figure 2.1: Examples of Neural Networks

2.2 激活函数

Equation 2.2表示的激活函数以阈值为界，一旦输入超过阈值，就切换输出。这样的函数称为“阶跃函数¹”。因此，可以说感知机中使用了阶跃函数作为激活函数。也就是说，在激活函数的众多候选函数中，感知机使用了阶跃函数。

2.2.1 sigmoid函数

神经网络中经常使用的一个激活函数是sigmoid函数（sigmoid function）：

$$h(x) = \frac{1}{1 + \exp(-x)} \quad (2.3)$$

2.2.2 sigmoid函数和阶跃函数的比较

首先注意到的是“平滑性”的不同。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。

接着说一下阶跃函数和sigmoid函数的共同性质。阶跃函数和sigmoid函数虽然在平滑性上有差异，可以发现它们具有相似的形状。实际上，两者的结构均是“输入小时，输出接近0（为0）；随着输入增大，输出向1靠近（变成1）”。也就是说，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。

¹阶跃函数是指一旦输入超过阈值，就切换输出的函数。应该有更严格的数学定义

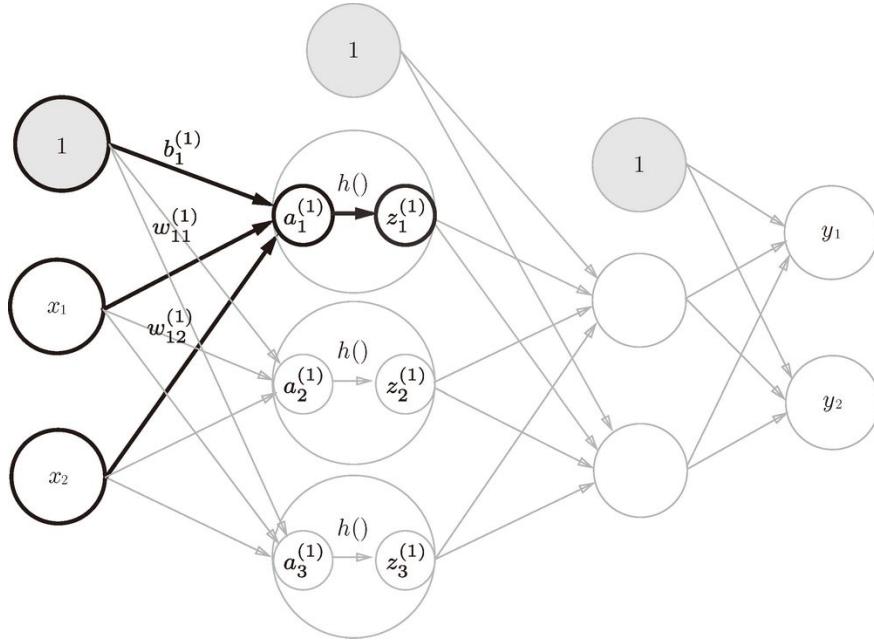


Figure 2.2: Signal passing from input layer to layer 1

2.2.3 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。

线性函数的问题在于，不管如何加深层数，总是存在与之等效的“无隐藏层的神经网络”。为了具体地（稍微直观地）理解这一点，我们来思考下面这个简单的例子。这里我们考虑把线性函数 $h(x) = cx$ 作为激活函数，把 $y(x) = h(h(h(x)))$ 的运算对应 3 层神经网络²。这个运算会进行 $y(x) = c \times c \times c \times x$ 的乘法运算，但是同样的处理可以由 $y(x) = ax$ （注意， $a = c^3$ ）这一次乘法运算（即没有隐藏层的神经网络）来表示。如本例所示，使用线性函数时，无法发挥多层网络带来的优势。因此，为了发挥叠加层所带来的优势，激活函数必须使用非线性函数。

2.2.4 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函数可以表示为：

$$h(x) = \begin{cases} x & , x > 0 \\ 0 & , x \leq 0 \end{cases} \quad (2.4)$$

2.3 多维数组的运算

`np.dot()`接收两个NumPy数组作为参数，并返回数组的乘积。

2.4 3层神经网络的实现

Figure 2.2中增加了表示偏置的神经元“1”。请注意，偏置的右下角的索引号只有一个。这是因为前一层的偏置神经元（神经元“1”）只有一个³。

²该对应只是一个近似，实际的神经网络运算比这个例子要复杂，但不影响后面的结论成立。

³任何前一层的偏置神经元“1”都只有一个。偏置权重的数量取决于后一层的神经元的数量（不包括后一层的偏置神经元“1”）。偏置是在后一层的神经元上增加，每个样本增加的偏置是相同，但是分量不一样。

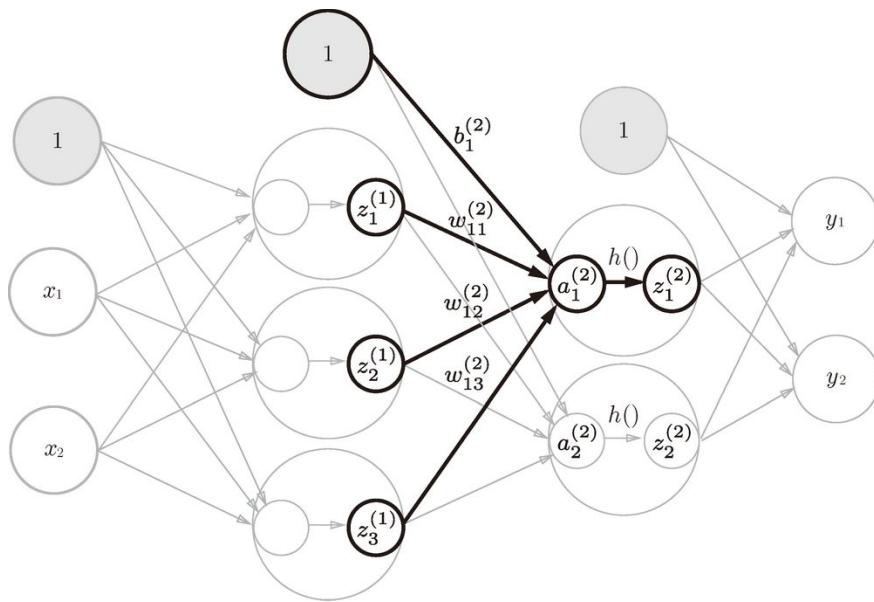


Figure 2.3: Layer 1 to Layer 2 Signaling

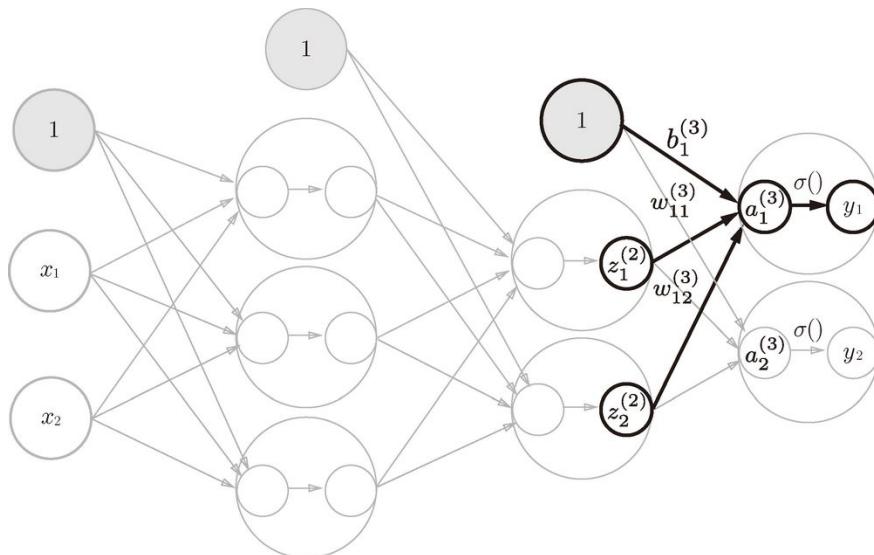


Figure 2.4: Signal passing from layer 2 to output layer

2.5 输出层的设计

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。

2.5.1 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。因此，在输出层使用恒等函数时，输入信号会原封不动地被输出。和前面介绍的隐藏层的激活函数一样，恒等函数进行的转换处理可以用一根箭头来表示。

分类问题中使用的softmax函数表示为：

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^n \exp(a_i)} \quad (2.5)$$

softmax函数的分子是输入信号 a_k 的指数函数，分母是所有输入信号的指数函数的和。

2.5.2 实现softmax函数时的注意事项

Equation 2.5在计算机的运算上有一定的缺陷。这个缺陷就是溢出问题。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。

$$\begin{aligned} y_k &= \frac{\exp(a_k)}{\sum_{i=1}^n \exp(a_i)} = \frac{C \exp(a_k)}{C \sum_{i=1}^n \exp(a_i)} \\ &= \frac{\exp(a_k + \log C)}{\sum_{i=1}^n \exp(a_i + \log C)} \\ &= \frac{\exp(a_k + C')}{\sum_{i=1}^n \exp(a_i + C')} \end{aligned} \quad (2.6)$$

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。这里的 C' 可以使用任何值，但是为了防止溢出，一般会使用输入信号中的最大值。

2.5.3 softmax函数的特征

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。

2.5.4 输出层的神经元数量

输出层的神经元数量需要根据待解决的问题来决定。对于分类问题，输出层的神经元数量一般设定为类别的数量。

2.6 手写数字识别

假设学习已经全部结束，我们使用学习到的参数，先实现神经网络的“推理处理”。这个推理处理也称为神经网络的前向传播（forward propagation）。

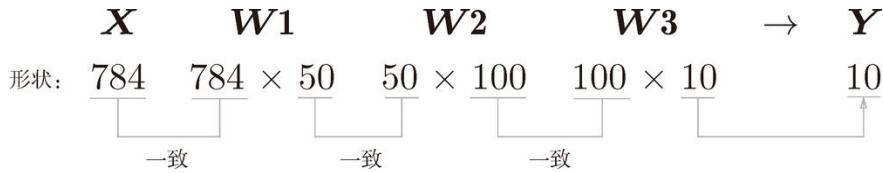


Figure 2.5: change of array shape

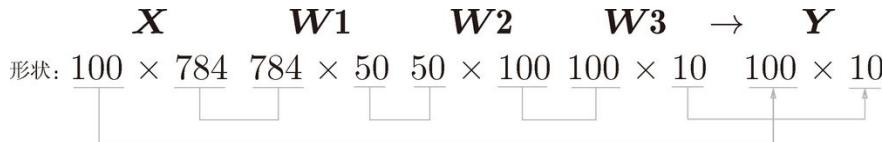


Figure 2.6: Variation of array shape in batch

2.6.1 批处理

从整体的处理流程来看, Figure 2.5中, 输入一个由784个元素(原本是一个 28×28 的二维数组)构成的一维数组后, 输出一个有10个元素的一维数组。这是只输入一张图像数据时的处理流程。

现在我们来考虑打包输入多张图像的情形。这种打包式的输入数据称为批(batch)。批有“捆”的意思, 图像就如同纸币一样扎成一捆。

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。

2.7 小结

- 神经网络中的激活函数使用平滑变化的sigmoid函数或ReLU函数。
- 通过巧妙地使用NumPy多维数组, 可以高效地实现神经网络。
- 机器学习的问题大体上可以分为回归问题和分类问题。
- 关于输出层的激活函数, 回归问题中一般用恒等函数, 分类问题中一般用softmax函数。
- 分类问题中, 输出层的神经元的数量设置为要分类的类别数。
- 输入数据的集合称为批。通过以批为单位进行推理处理, 能够实现高速的运算。

Chapter 3

神经网络的学习

本章的主题是神经网络的学习。这里所说的“学习”是指从训练数据中自动获取最优权重参数的过程。

3.1 从数据中学习

神经网络的特征就是可以从数据中学习。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。

3.1.1 数据驱动

如果让我们自己来设计一个能将5正确分类的程序，就会意外地发现这是一个很难的问题。人可以简单地识别出5，但却很难明确说出是基于何种规律而识别出了5。

深度学习有时也称为端到端机器学习（end-to-end machine learning）。这里所说的端到端是指从一端到另一端的意思，也就是从原始数据（输入）中获得目标结果（输出）的意思。

3.1.2 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。为什么需要



Figure 3.1: Moving from human designed rules to machine learning from data

将数据分为训练数据和测试数据呢？因为我们追求的是模型的泛化能力。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。

3.2 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。

损失函数是表示神经网络性能的“恶劣程度”的指标，即当前的神经网络对监督数据在多大程度上不拟合，在多大程度上不一致。以“性能的恶劣程度”为指标可能会使人感到不太自然，但是如果给损失函数乘上一个负值，就可以解释为“在多大程度上不坏”，即“性能有多好”。并且，“使性能的恶劣程度达到最小”和“使性能的优良程度达到最大”是等价的，不管是用“恶劣程度”还是“优良程度”，做的事情本质上都是一样的。

3.2.1 均方误差

可以用作损失函数的函数有很多，其中最有名的是均方误差（mean squared error）。均方误差如下式所示。

$$E = \frac{1}{2} \sum_k (y_k - t_k)^2 \quad (3.1)$$

这里， y_k 是表示神经网络的输出， t_k 表示监督数据， k 表示数据的维数。

3.2.2 交叉熵误差

除了均方误差之外，交叉熵误差（cross entropy error）也经常被用作损失函数。交叉熵误差如下式所示：

$$E = - \sum_k t_k \log y_k \quad (3.2)$$

3.2.3 mini-batch学习

机器学习使用训练数据进行学习。使用训练数据进行学习，严格来说，就是针对训练数据计算损失函数的值，找出使该值尽可能小的参数。因此，计算损失函数时必须将所有的训练数据作为对象。

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，Equation 3.2改写为：

$$E = - \frac{1}{N} \sum_n \sum_k t_{nk} \log y_{nk} \quad (3.3)$$

式中，假设数据有 N 个， t_{nk} 表示第 n 个数据的第 k 个元素的值（ y_{nk} 是神经网络的输出， t_{nk} 是监督数据）。

如果遇到大数据，数据量会有几百万、几千万之多，这种情况下以全部数据为对象计算损失函数是不现实的。因此，我们从全部数据中选出一部分，作为全部数据的“近似”。神经网络的学习也是从训练数据中选出一批数据（称为mini-batch,小批量），然后对每个mini-batch进行学习。这种学习方式称为**mini-batch学习**。

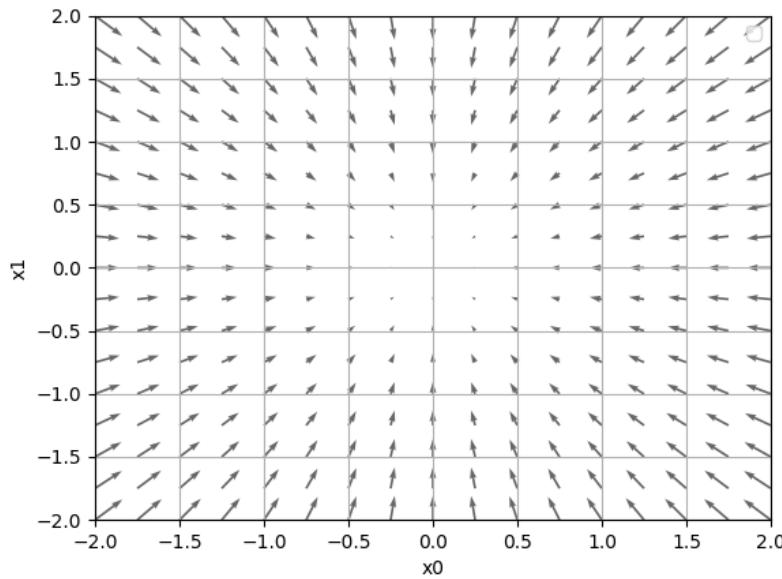


Figure 3.2: 2D Gradient Schematic

计算电视收视率时，并不会统计所有家庭的电视机，而是仅以那些被选中的家庭为统计对象。比如，通过从关东地区随机选择1000个家庭计算收视率，可以近似地求得关东地区整体的收视率。这1000个家庭的收视率，虽然严格上不等于整体的收视率，但可以作为整体的一个近似值。和收视率一样，mini-batch的损失函数也是利用一部分样本数据来近似地计算整体。也就是说，用随机选择的小批量数据（mini-batch）作为全体训练数据的近似值。

3.2.4 mini-batch版交叉熵误差的实现

3.2.5 为何要设定损失函数

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。

其实我觉得是数学层面的妥协，因为没有办法给出精度的准备数学方程，因此很多求解方法都无法应用。

3.3 数值微分

3.4 梯度

像 $\left(\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}\right)$ 这样的由全部变量的偏导数汇总而成的向量称为梯度（gradient）。

实际上，梯度会指向各点处的函数值降低的方向。更严格地讲，梯度指示的方向是各点处的函数值减小最多的方向¹。这是一个非常重要的性质。

¹Directional derivative = $\cos(\theta) \times \text{grad}$ (θ 是方向导数的方向与梯度方向的夹角)。因此，所有的下降方向中，梯度方向下降最多。

3.4.1 梯度法

机器学习的主要任务是在学习时寻找最优参数。同样地，神经网络也必须在学习时找到最优参数（权重和偏置）。这里所说的最优参数是指损失函数取最小值时的参数。但是，一般而言，损失函数很复杂，参数空间庞大，我们不知道它在何处能取得最小值。而通过巧妙地使用梯度来寻找函数最小值（或者尽可能小的值）的方法就是梯度法。

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。

函数的极小值、最小值以及被称为鞍点（saddle point）的地方，梯度为0。极小值是局部最小值，也就是限定在某个范围内的最小值。鞍点是从某个方向上看是极大值，从另一个方向上看则是极小值的点。虽然梯度法是要寻找梯度为0的地方，但是那个地方不一定就是最小值（也有可能是极小值或者鞍点）。此外，当函数很复杂且呈扁平状时，学习可能会进入一个（几乎）平坦的地区，陷入被称为“学习高原”的无法前进的停滞期。

虽然梯度的方向并不一定指向最小值，但沿着它的方向能够最大限度地减小函数的值。因此，在寻找函数的最小值（或者尽可能小的值）的位置的任务中，要以梯度的信息为线索，决定前进的方向。

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。

根据目的是寻找最小值还是最大值，梯度法的叫法有所不同。严格地讲，寻找最小值的梯度法称为梯度下降法（gradient descent method），寻找最大值的梯度法称为梯度上升法（gradient ascent method）。但是通过反转损失函数的符号，求最小值的问题和求最大值的问题会变成相同的问题，因此“下降”还是“上升”的差异本质上并不重要。一般来说，神经网络（深度学习）中，梯度法主要是指梯度下降法。

用数学式来表示梯度法：

$$\begin{aligned}x_0 &= x_0 - \eta \frac{\partial f}{\partial x_0} \\x_1 &= x_1 - \eta \frac{\partial f}{\partial x_1}\end{aligned}\tag{3.4}$$

式中 η 表示更新量，在神经网络的学习中，称为学习率（learning rate）。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。学习率需要事先确定为某个值，比如0.01或0.001。一般而言，这个值过大或过小，都无法抵达一个“好的位置”。在神经网络的学习中，一般会一边改变学习率的值，一边确认学习是否正确进行了。

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

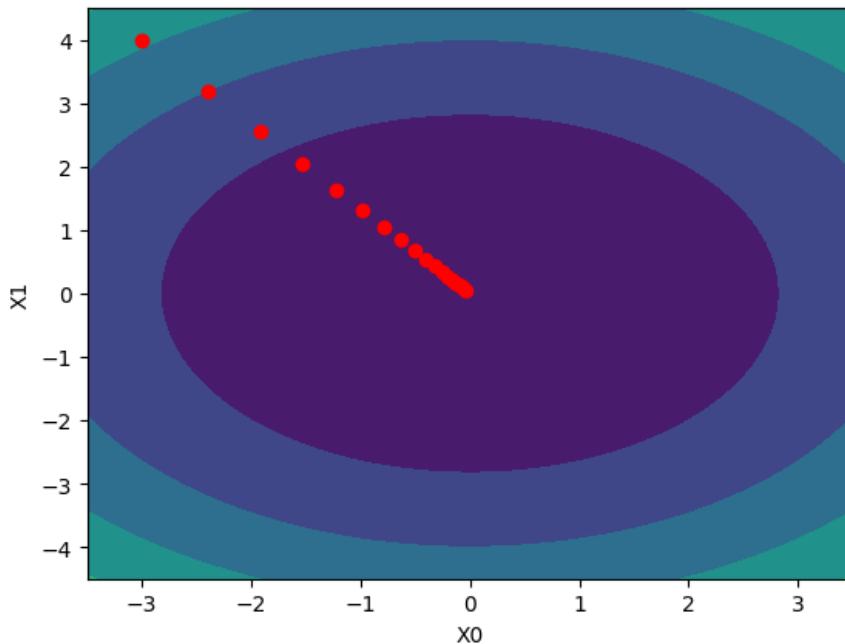


Figure 3.3: The update process of the gradient method

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。

3.4.2 神经网络的梯度

3.5 学习算法的实现

神经网络的学习步骤如下所示：

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”。

- 步骤1（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。
- 步骤2（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。
- 步骤3（更新参数）将权重参数沿梯度方向进行微小更新。
- 步骤4（重复）重复步骤1、步骤2、步骤3。

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”指的是“随机选择的”的意思，因此，随机梯度下降法是“对随机选择的数据进行的梯度下降法”。深度学习的很多框架中，随机梯度下降法一般由一个名为SGD的函数来实现。

3.5.1 基于测试数据的评价

我们确认了通过反复学习可以使损失函数的值逐渐减小这一事实。不过这个损失函数的值，严格地讲是“对训练数据的某个mini-batch的损失函数”的值。训练数据的损失函数值减小，虽说是神经网络的学习正常进行的一个信号，但光看这个结果还不能说明该神经网络在其他数据集上也一定能有同等程度的表现。

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。

神经网络学习的最初目标是掌握泛化能力，因此，要评价神经网络的泛化能力，就必须使用不包含在训练数据中的数据。

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。

Chapter 4

误差反向传播法

通过数值微分计算了神经网络的权重参数的梯度（严格来说，是损失函数关于权重参数的梯度）。数值微分虽然简单，也容易实现，但缺点是计算上比较费时间。本章我们将学习一个能够高效计算权重参数的梯度的方法——误差反向传播法。

4.1 计算图

计算图将计算过程用图形表示出来。这里说的图形是数据结构图，通过多个节点和边表示（连接节点的直线称为“边”）。

计算图通过节点和箭头表示计算过程。节点用○表示，○中是计算的内容。将计算的中间结果写在箭头的上方，表示各个节点的计算结果从左向右传递。

虽然Figure 4.1中把“ $\times 2$ ”“ $\times 1.1$ ”等作为一个运算整体用○括起来了，不过只用○表示乘法运算“ \times ”也是可行的。此时，如Figure 4.2所示，可以将“2”和“1.1”分别作为变量“苹果的个数”和“消费税”标在○外面。

用计算图解题的情况下，需要按如下流程进行。

1. 构建计算图。
2. 在计算图上，从左向右进行计算。

这里的第2步“从左向右进行计算”是一种正方向上的传播，简称为正向传播（forward propagation）。正向传播是从计算图出发点到结束点的传播。既然有正向传播这个名称，当然也可以考虑反向（从图上看的话，就是从右向左）的传播。实际上，这种传播称为反向传播（backward propagation）。反向传播将在接下来的导数计算中发挥重要作用。

4.1.1 局部计算

计算图的特征是可以通过传递“局部计算”获得最终结果。“局部”这个词的意思是“与自己相关的某个小范围”。局部计算是指，无论全局发生了什么，都能只根据与自己相关的信息输出接下来的结果。

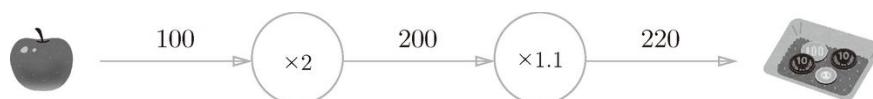


Figure 4.1: Based on the calculation graph to solve the answer

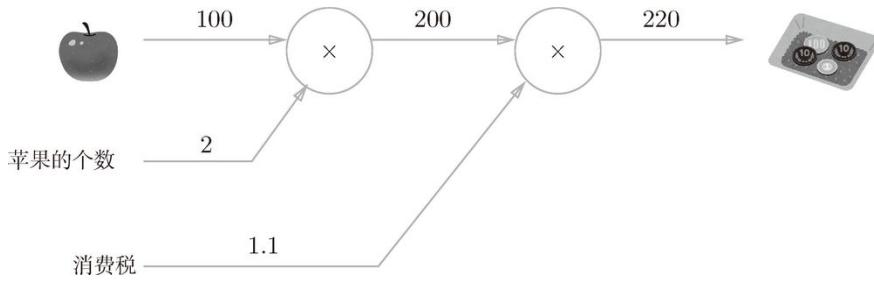


Figure 4.2: Based on the calculation graph to solve the answer2

4.2 链式法则

前面介绍的计算图的正向传播将计算结果正向（从左到右）传递，其计算过程是我们日常接触的计算过程，所以感觉上可能比较自然。而反向传播将局部导数向正方向的反方向（从右到左）传递，一开始可能会让人感到困惑。传递这个局部导数的原理，是基于链式法则（chain rule）的。

4.2.1 计算图的反向传播

4.2.2 什么是链式法则

介绍链式法则时，我们需要先从复合函数说起。复合函数是由多个函数构成的函数。比如， $z = (x+y)^2$ 是下面所示的两个式子构成的：

$$\begin{aligned} z &= t^2 \\ t &= x + y \end{aligned} \tag{4.1}$$

链式法则是关于复合函数的导数的性质，定义如下：

如果某个函数由复合函数表示，则该复合函数的导数可以用构成复合函数的各个函数的导数的乘积表示。

以Equation 4.1为例， $\frac{\partial z}{\partial x}$ 可以用 $\frac{\partial z}{\partial t}$ 和 $\frac{\partial t}{\partial x}$ 的乘积表示，可以写成下式：

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t} \frac{\partial t}{\partial x} \tag{4.2}$$

对于 $z = (x+y)^2$ ，那么就有

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t} \frac{\partial t}{\partial x} = 2t * 1 = 2(x+y)$$

4.2.3 链式法则和计算图

如图所示，计算图的反向传播从右到左传播信号。反向传播的计算顺序是，先将节点的输入信号乘以节点的局部导数（偏导数），然后再传递给下一个节点。比如，反向传播时，“**2”节点的输入是 $\frac{\partial z}{\partial t}$ ，将其乘以局部导数 $\frac{\partial z}{\partial t}$ （因为正向传播时输入是 t 、输出是 z ，所以这个节点的局部导数是 $\frac{\partial z}{\partial t}$ ），然后传递给下一个节点。另外，图5-7中反向传播最开始的信号 $\frac{\partial z}{\partial t}$ 在前面的数学式中没有出现，这是因为 $\frac{\partial z}{\partial t} = 1$ ，所以在刚才的式子中被省略了。

4.3 反向传播

上一节介绍了计算图的反向传播是基于链式法则成立的。本节将以“+”和“×”等运算为例，介绍反向传播的结构。

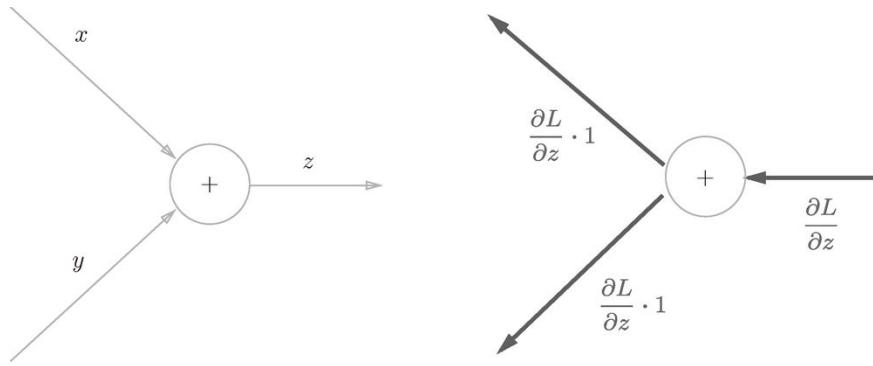


Figure 4.3: Backpropagation for Adder nodes

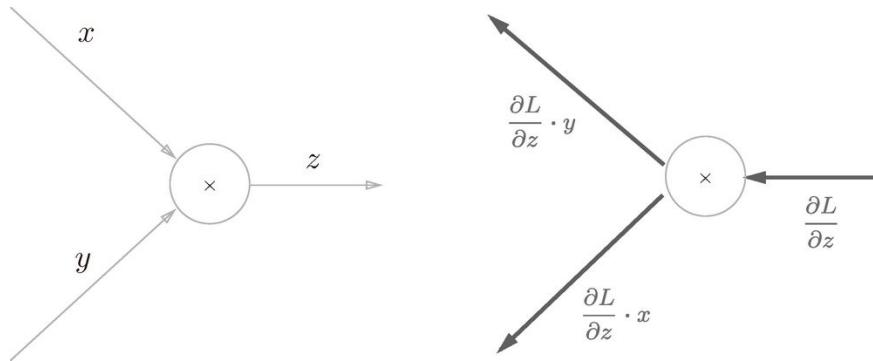


Figure 4.4: Backpropagation of multiply nodes

4.3.1 加法节点的反向传播

考虑加法节点的反向传播。这里以 $z = x + y$ 为对象，观察它的反向传播。 $z = x + y$ 的导数可由下式（解析性地）计算出来：

$$\begin{aligned}\frac{\partial z}{\partial x} &= 1 \\ \frac{\partial z}{\partial y} &= 1\end{aligned}$$

Figure 4.3, 反向传播将从上游传过来的导数乘以1，然后传向下游。也就是说，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。

4.3.2 乘法节点的反向传播

看一下乘法节点的反向传播。这里我们考虑 $z = xy$ 。这个式子的导数用下式表示：

$$\begin{aligned}\frac{\partial z}{\partial x} &= y \\ \frac{\partial z}{\partial y} &= x\end{aligned}$$

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。

加法的反向传播只是将上游的值传给下游，并不需要正向传播的输入信号。但是，乘法的反向传播需要正向传播时的输入信号值。因此，实现乘法节点的反向传播时，要保存正向传播的输入信号。

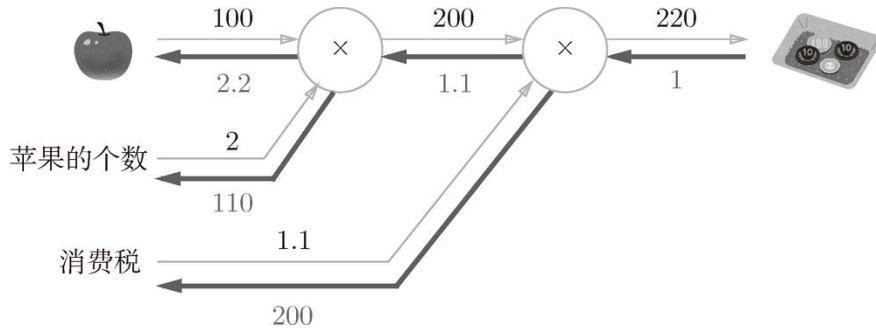


Figure 4.5: Example of backpropagation for buying apples

4.4 简单层的实现

我们把要实现的计算图的乘法节点称为“乘法层”（MulLayer），加法节点称为“加法层”（AddLayer）。

4.4.1 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。

4.5 激活函数层的实现

现在，我们将计算图的思路应用到神经网络中。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。

4.5.1 ReLU层

激活函数ReLU（Rectified Linear Unit）由下式表示：

$$y = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

可以求出y关于x的导数，如下式所示：

$$\frac{\partial y}{\partial x} = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases} \quad (4.3)$$

Equation 4.3中，如果正向传播时的输入x大于0，则反向传播会将上游的值原封不动地传给下游。反过来，如果正向传播时的x小于等于0，则反向传播中传给下游的信号将停在此处。

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。

4.5.2 Sigmoid层

sigmoid函数由下式表示：

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

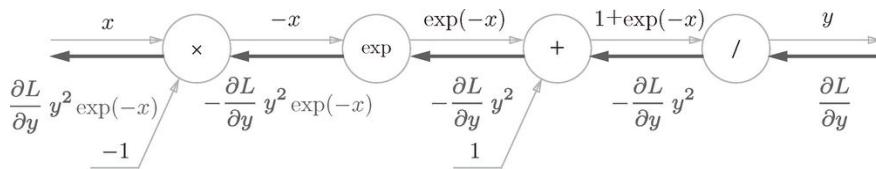


Figure 4.6: Computational graph of the Sigmoid layer

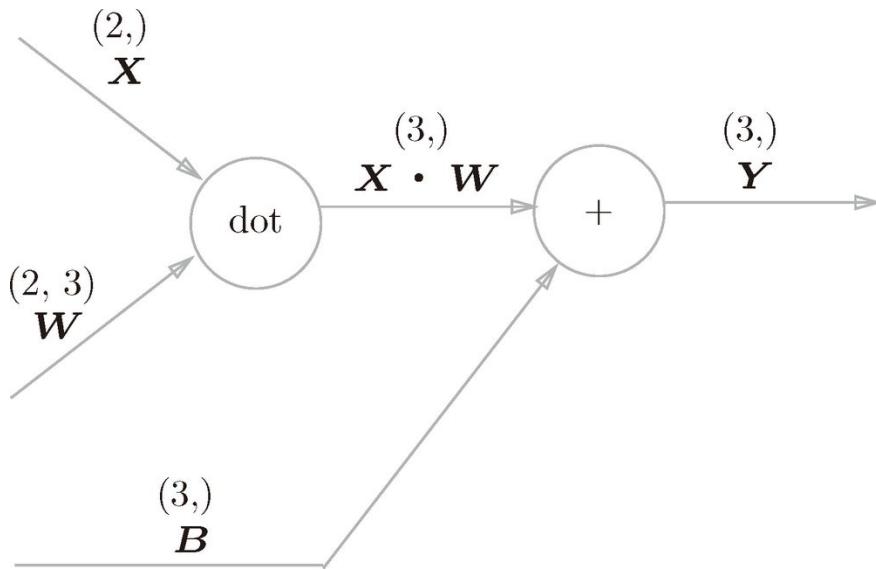


Figure 4.7: Computational graph of the Affine layer

另外， $\frac{\partial L}{\partial y} y^2 \exp(-x)$ 可以进一步整理如下：

$$\begin{aligned}\frac{\partial L}{\partial y} y^2 \exp(-x) &= \frac{\partial L}{\partial y} \frac{1}{(1 + \exp(-x))^2} \exp(-x) \\ &= \frac{\partial L}{\partial y} \frac{1}{1 + \exp(-x)} \frac{\exp(-x)}{1 + \exp(-x)} \\ &= \frac{\partial L}{\partial y} y(1 - y)\end{aligned}$$

4.6 Affine/Softmax层的实现

4.6.1 Affine层

神经元的加权和可以用 $Y = np.dot(X, W) + B$ 计算出来。然后， Y 经过激活函数转换后，传递给下一层。这就是神经网络正向传播的流程。

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”^a。因此，这里将进行仿射变换的处理实现为“Affine层”。

^a 几何中，仿射变换包括一次线性变换和一次平移，分别对应神经网络的加权和运算与加偏置运算。

Figure 4.7展示了Affine层的计算图（注意变量是矩阵，各个变量的上方标记了该变量的形状）

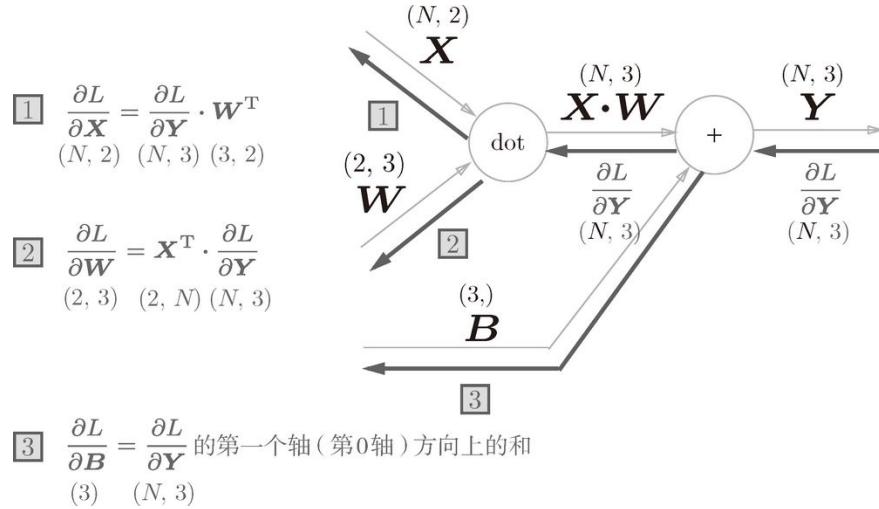


Figure 4.8: Computational graph of the batch version of the Affine layer

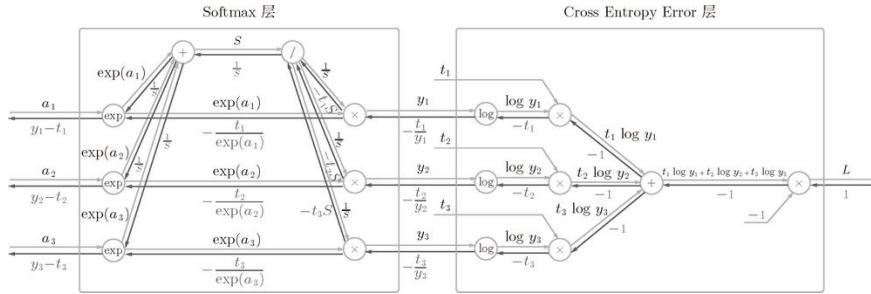


Figure 4.9: Computational graph of the Softmax-with-Loss layer

以矩阵为对象的反向传播，按矩阵的各个元素进行计算时，步骤和以标量为对象的计算图相同。实际写一下的话，可以得到下式

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{X}} &= \frac{\partial L}{\partial \mathbf{Y}} \mathbf{W}^T \\ \frac{\partial L}{\partial \mathbf{W}} &= \mathbf{X}^T \frac{\partial L}{\partial \mathbf{Y}}\end{aligned}\tag{4.4}$$

4.6.2 批版本的Affine层

前面介绍的Affine层的输入 \mathbf{X} 是以单个数据为对象的。现在我们考虑 N 个数据一起进行正向传播的情况，也就是批版本的Affine层。

正向传播时，偏置会被加到每一个数据（第1个、第2个……）上。因此，反向传播时，各个数据的反向传播的值需要汇总为偏置的元素。

4.6.3 Softmax-with-Loss 层

softmax函数会将输入值正规化之后再输出。

考虑到这里也包含作为损失函数的交叉熵误差（cross entropy error），所以称为“Softmax-with-Loss层”。Softmax-with- Loss层（Softmax函数和交叉熵误差）的计算图如Figure 4.9所示。

反向传播的具体的推导参见Appendix A

Figure 4.10计算图中，softmax函数记为Softmax层，交叉熵误差记为Cross Entropy Error层。这里假设要进行3类分类，从前面的层接收3个输入（得分）。如Figure 4.10所示，Softmax层将输入 (a_1, a_2, a_3) 正

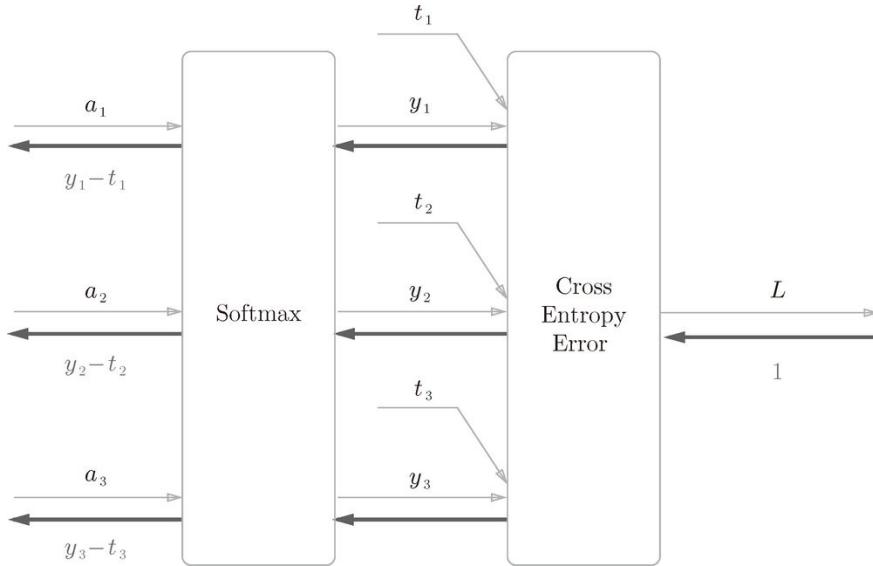


Figure 4.10: Simple version of the calculation graph of the Softmax-with-Loss layer

规范化，输出 (y_1, y_2, y_3) 。Cross Entropy Error层接收Softmax的输出 (y_1, y_2, y_3) 和训练标签 (t_1, t_2, t_3) ，从这些数据中输出损失 L 。

Softmax层的反向传播得到了 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$ 这样“漂亮”的结果。由于 y_1, y_2, y_3 是Softmax层的输出， (t_1, t_2, t_3) 是监督数据，所以 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$ 是Softmax层的输出和训练标签的差分。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。

神经网络学习的目的就是通过调整权重参数，使神经网络的输出（Softmax 的输出）接近训练标签。因此，必须将神经网络的输出与训练标签的误差高效地传递给前面的层。刚刚的 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$ 正是Softmax层的输出与训练标签的差，直截了当地表示了当前神经网络的输出与训练标签的误差。

好的损失函数的意义

使用交叉熵误差作为 softmax 函数的损失函数后，反向传播得到 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$ 这样“漂亮”的结果。实际上，这样“漂亮”的结果并不是偶然的，而是为了得到这样的结果，特意设计了交叉熵误差函数。回归问题中输出层使用“恒等函数”，损失函数使用“平方和误差”，也是出于同样的理由。也就是说，使用“平方和误差”作为“恒等函数”的损失函数，反向传播才能得到 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$ 这样“漂亮”的结果。

4.7 误差反向传播法的实现

数值微分虽然实现简单，但是计算要耗费较多的时间。和需要花费较多时间的数值微分不同，误差反向传播法可以快速高效地计算梯度。

4.7.1 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。因此，后文将不再使用耗费时间的数值微分，而是使用误差反向传播法求梯度。

数值微分的计算很耗费时间，而且如果有误差反向传播法的（正确的）实现的话，就没有必要使用数值微分的实现了。那么数值微分有什么用呢？

数值微分的优点是实现简单，因此，一般情况下不太容易出错。而误差反向传播法的实现很复杂，容易出错。所以，经常会比较数值微分的结果和误差反向传播法的结果，以确认误差反向传播法的实现是否正确。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）。

4.7.2 使用误差反向传播法的学习

Chapter 5

与学习相关的技巧

本章将介绍神经网络的学习中的一些重要观点，主题涉及寻找最优权重参数的最优化方法、权重参数的初始值、超参数的设定方法等。此外，为了应对过拟合，本章还将介绍权值衰减、Dropout等正则化方法，并进行实现。

5.1 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）。遗憾的是，神经网络的最优化问题非常难。这是因为参数空间非常复杂，无法轻易找到最优解（无法使用那种通过解数学式一下子就求得最小值的方法）。而且，在深度神经网络中，参数的数量非常庞大，导致最优化问题更加复杂。

使用参数的梯度，沿梯度方向更新参数，并重复这个步骤多次，从而逐渐靠近最优参数，这个过程称为**随机梯度下降法**（stochastic gradient descent），简称**SGD**。

5.1.1 SGD

用数学式可以将SGD写成如下式：

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

SGD是朝着梯度方向只前进一定距离的简单方法。

5.1.2 SGD的缺点

虽然SGD简单，并且容易实现，但是在解决某些问题时可能没有效率。

$$z = \frac{1}{20}x^2 + y^2$$

上式表示的函数是向 x 轴方向延伸的“碗”状函数。

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。**

5.1.3 Momentum

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}} \quad (5.1a)$$

$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v} \quad (5.1b)$$



Figure 5.1: Momentum-The ball rolls on an incline

这里新出现了一个变量 v , 对应物理上的速度。Equation 5.1a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。如Figure 5.1所示, Momentum方法给人的感觉就像是小球在地面上滚动。

式Equation 5.1a中有 αv 这一项。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之类的价值), 对应物理上的地面摩擦或空气阻力。

5.1.4 AdaGrad

在关于学习率的有效技巧中, 有一种被称为学习率衰减 (learning rate decay) 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。

逐渐减小学习率的想法, 相当于将“全体”参数的学习率值一起降低。而AdaGrad进一步发展了这个想法, 针对“一个一个”的参数, 赋予其“定制”的值。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。

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}} \quad (5.2a)$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}} \quad (5.2b)$$

式中, \odot 表示对应矩阵元素的乘法, 在更新参数时, 通过乘以 $\frac{1}{\sqrt{h}}$, 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。

5.1.5 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。

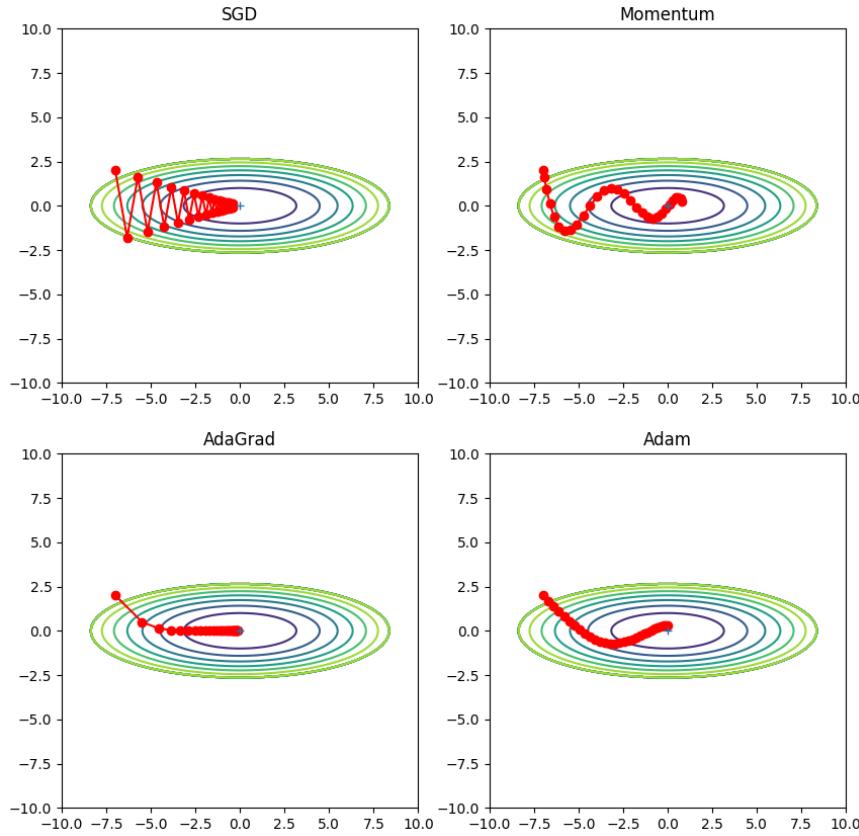


Figure 5.2: Comparison of Optimization Methods

5.1.6 使用哪种更新方法呢

如Figure 5.2所示，根据使用的方法不同，参数更新的路径也不同。只看这个图的话，AdaGrad似乎是最好的，不过也要注意，结果会根据要解决的问题而变。并且，很显然，超参数（学习率等）的设定值不同，结果也会发生变化。

非常遗憾，（目前）并不存在能在所有问题中都表现良好的方法。这4种方法各有各的特点，都有各自擅长解决的问题和不擅长解决的问题。

5.1.7 基于MNIST数据集的更新方法的比较

5.2 权重的初始值

在神经网络的学习中，权重的初始值特别重要。实际上，设定什么样的权重初始值，经常关系到神经网络的学习能否成功。

5.2.1 可以将权重初始值设为0吗

权值衰减(weights decay)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。从结论来说，将权重初始值设为0不是一个好主意。事实上，将权重初始值设为0的话，将无法正确进行学习。

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。

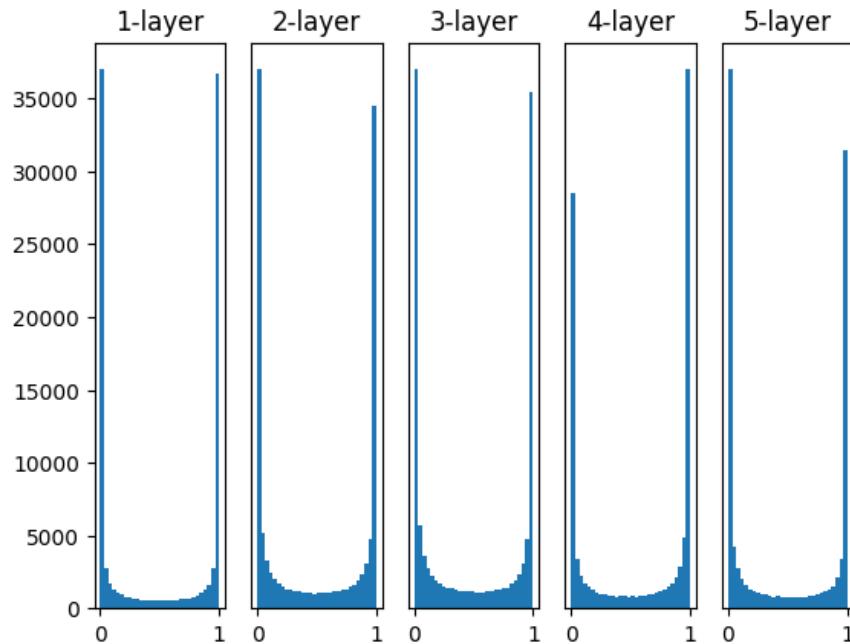


Figure 5.3: The distribution of the activation value of each layer when using a Gaussian distribution with a standard deviation of 1 as the initial weight value

5.2.2 隐藏层的激活值的分布

观察隐藏层的激活值¹（激活函数的输出数据）的分布，可以获得很多启发。从Figure 5.3可知，各层的激活值呈偏向0和1的分布。这里使用的sigmoid 函数是S型函数，随着输出不断地靠近0（或者靠近1），它的导数的值逐渐接近0。因此，偏向0和1的数据分布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。

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。

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。推导出的结论是，如果前一层的节点数为 n ，则初始值使用标准差为 $\sqrt{\frac{1}{n}}$ 的分布。

使用Xavier初始值后的结果如Figure 5.5所示。从这个结果可知，越是后面的层，图像变得越歪斜，但是呈现了比之前更有广度的分布。因为各层间传递的数据有适当的广度，所以sigmoid函数的表现力不受限制，有望进行高效的学习。

Figure 5.5的分布中，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

¹这里我们将激活函数的输出数据称为“激活值”，但是有的文献中会将在层之间流动的数据也称为“激活值”。

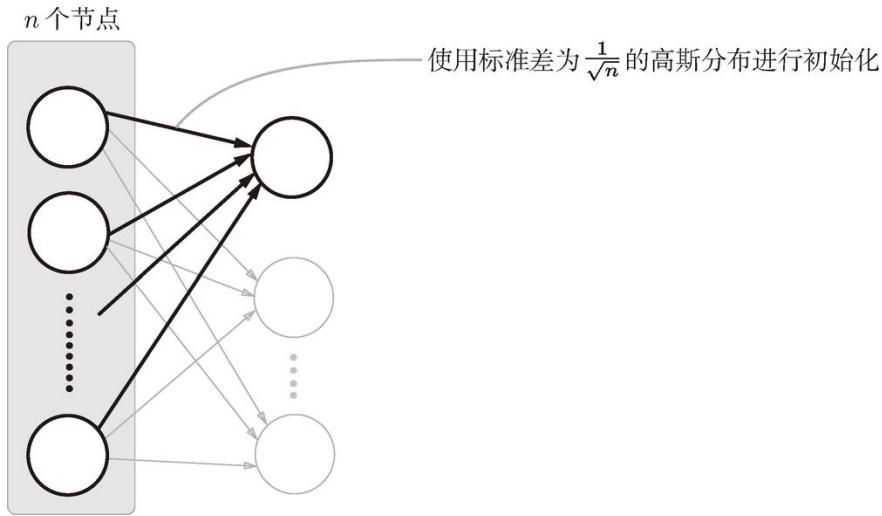


Figure 5.4: Xavier initial value

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。

5.2.3 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 $\sqrt{\frac{2}{n}}$ 的高斯分布。

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。

5.2.4 基于MNIST数据集的权重初始值的比较

这个实验中，神经网络有 5 层，每层有 100 个神经元，激活函数使用的是 ReLU。从 Figure 5.6 的结果可知， $std = 0.01$ 时完全无法进行学习。这和刚才观察到的激活值的分布一样，是因为正向传播中传递的值很小（集中在 0 附近的数据）。因此，逆向传播时求到的梯度也很小，权重几乎不进行更新。相反，当权重初始值为 Xavier 初始值和 He 初始值时，学习进行得很顺利。并且，我们发现 He 初始值时的学习进度更快一些。

5.3 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。

5.3.1 Batch Normalization 的算法

什么 Batch Norm 这么惹人注目呢？因为 Batch Norm 有以下优点。

- 可以使学习快速进行（可以增大学习率）。

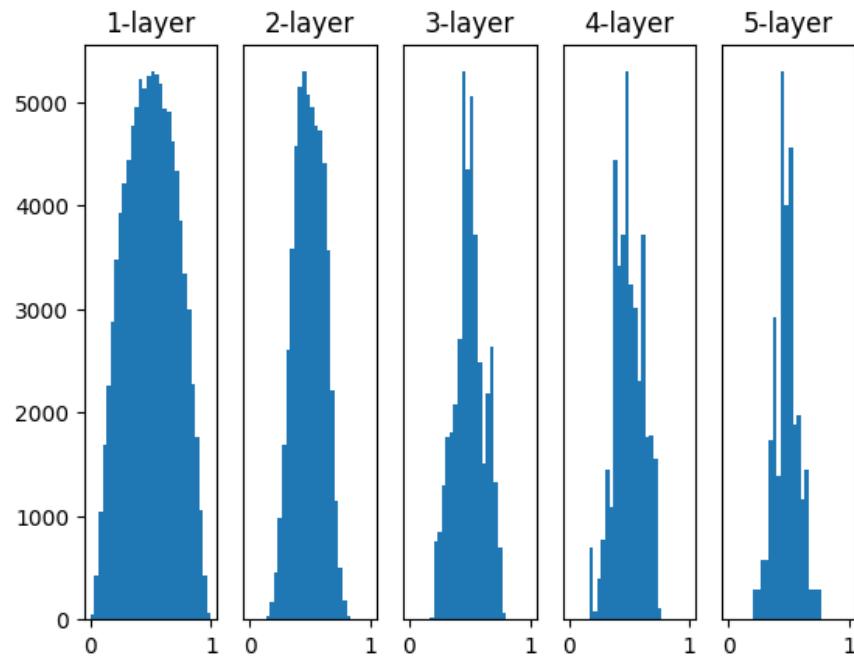


Figure 5.5: Distribution of activation values of each layer when using Xavier initial value as weight initial value

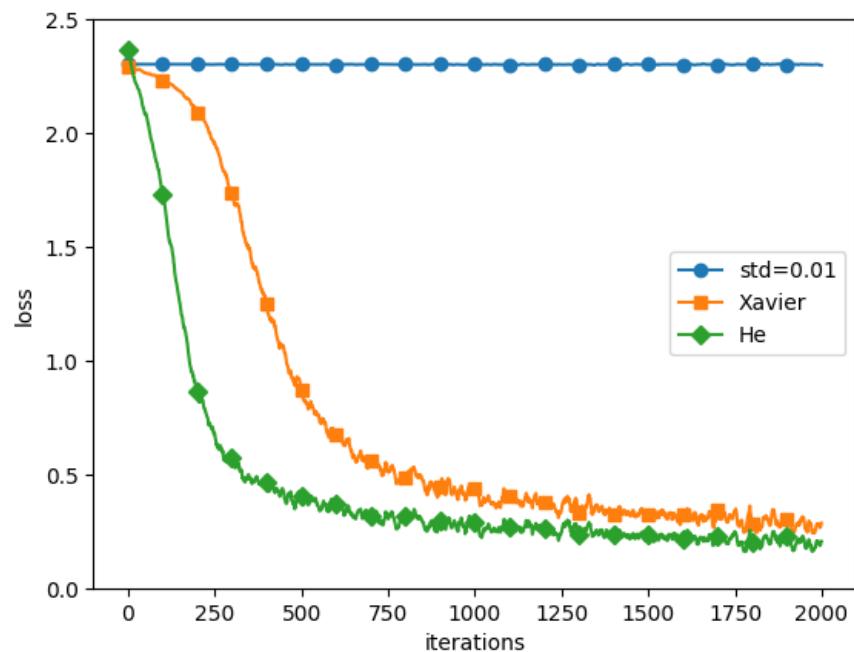


Figure 5.6: Comparison of weight initial values based on MNIST dataset

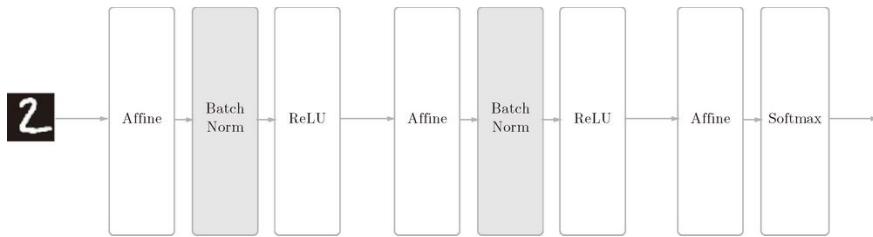


Figure 5.7: An example of a neural network using Batch Normalization

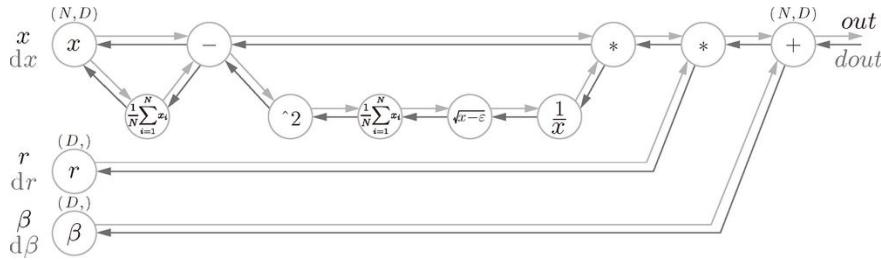


Figure 5.8: Computational graph of Batch Normalization

- 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）。

考虑到深度学习要花费很多时间，第一个优点令人非常开心。另外，后两点也可以帮我们消除深度学习中的很多烦恼。

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Batch Normalization层（下文简称Batch Norm层），如Figure 5.7所示。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：

$$\begin{aligned}\mu_B &\leftarrow \frac{1}{m} \sum_{i=1}^m x_i \\ \sigma_B^2 &\leftarrow \frac{1}{m-1} \sum_{i=1}^m (x_i - \mu_B)^2 \\ \hat{x}_i &\leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}\end{aligned}$$

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：

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

这里， γ 和 β 是参数。一开始 $\gamma = 1$, $\beta = 0$ ，然后再通过学习调整到合适的值。

如果使用Figure 5.8的计算图来思考的话，Batch Norm的反向传播或许也能比较轻松地推导出来。Frederik Kratzert 的博客“[Understanding the backward pass through Batch Normalization Layer](#)”里有详细说明。

5.3.2 Batch Normalization的评估

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。

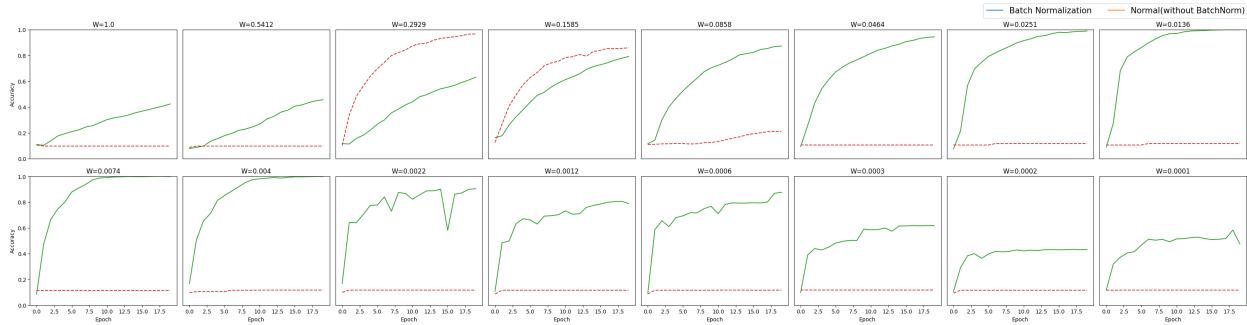


Figure 5.9: Use batch norm to compare with no use

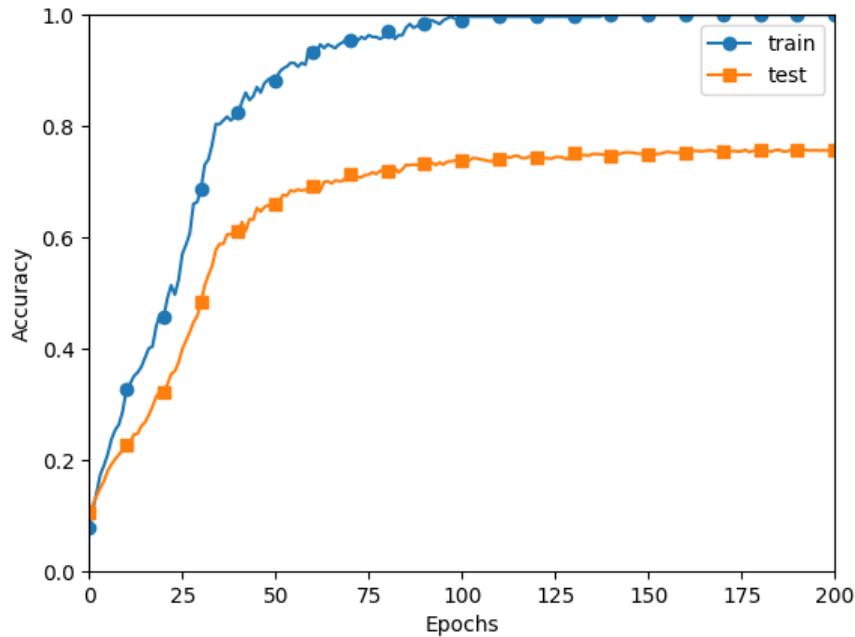


Figure 5.10: Comparison of recognition accuracy between training data and test data

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）。

5.4 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，但是相应地，抑制过拟合的技巧也很重要。

5.4.1 过拟合

发生过拟合的原因，主要有以下两个：

1. 模型拥有大量参数、表现力强。
2. 训练数据少。

Figure 5.10 中，使用的数据量仅为300，神经网络的层数为7，这是故意满足过拟合条件的情况。

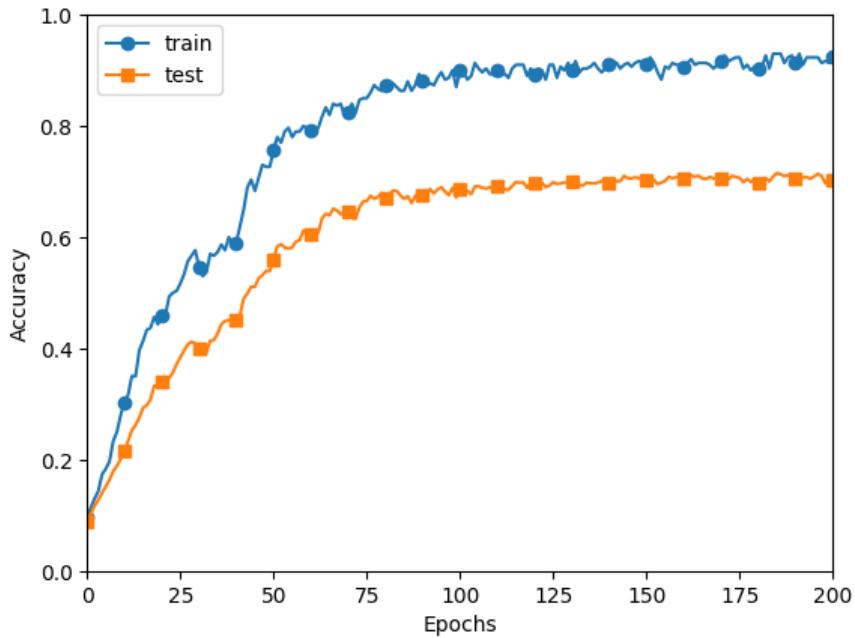


Figure 5.11: Changes in recognition accuracy of training data and test data using weight decay

5.4.2 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。

L_2 范数相当于各个元素的平方和。用数学式表示的话，假设有权重 $W = (w_1, w_2, \dots, w_n)$ ，则 L_2 范数可用计算 $\sqrt{w_1^2 + w_2^2 + \dots + w_n^2}$ 出来。除了 L_2 范数，还有 L_1 范数、 L_∞ 范数等。 L_1 范数是各个元素的绝对值之和，相当于 $|w_1| + |w_2| + \dots + |w_n|$ 。 L_∞ 范数也称为 Max 范数，相当于各个元素的绝对值中最大的那一个。 L_2 范数、 L_1 范数、 L_∞ 范数都可以用作正则化项，它们各有各的特点，不过这里我们要实现的是比较常用的 L_2 范数。

Figure 5.11 使用了权值衰退，这减少了过拟合现象，但是代价是降低了训练集的准确率，但是测试集的准确率没有任何的提升，也就是说这里的防止过拟合只是抑制训练集精度，模型似乎还是不是一个好的模型。

5.4.3 Dropout

作为抑制过拟合的方法，为损失函数加上权重的 L_2 范数的权值衰减方法。该方法可以简单地实现，在某种程度上能够抑制过拟合。但是，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，如 Figure 5.12 所示。训练时，每传递一次数据，就会随机选择要删除的神经元。然后，测试时，虽然会传递所有的神经元信号，但是对于各个神经元的输出，要乘上训练时的删除比例后再输出。

Figure 5.13 中，通过使用 Dropout，训练数据和测试数据的识别精度的差距变小了。并且，训练数

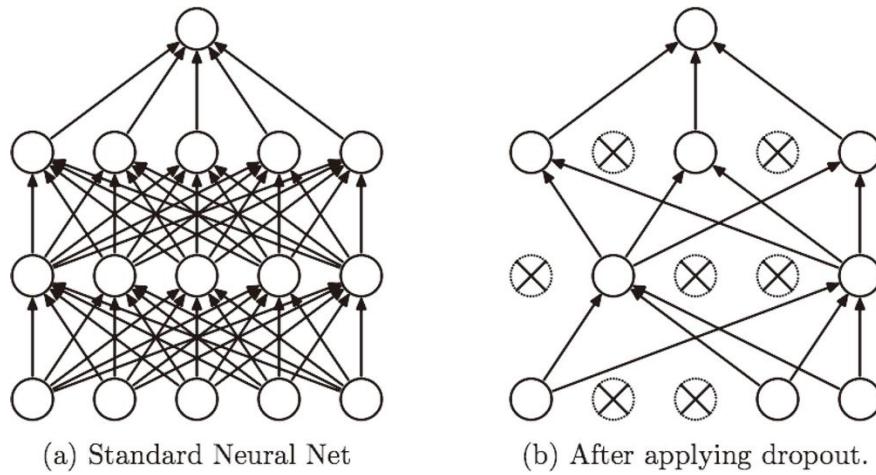


Figure 5.12: Concept map of Dropout

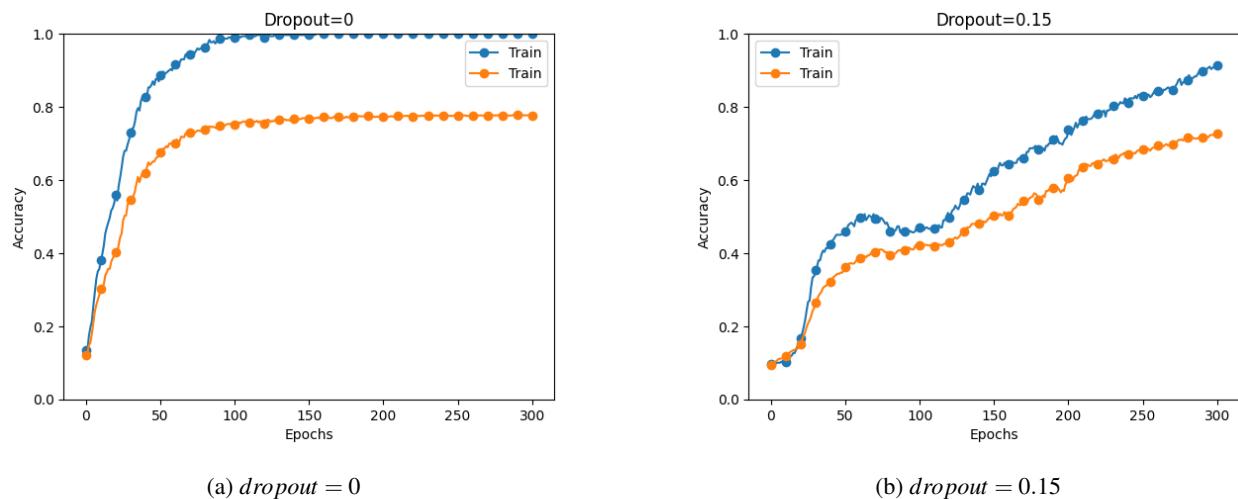


Figure 5.13: Comparison between dropout use or not

据也没有到达100%的识别精度。像这样，通过使用 Dropout，即便是表现力强的网络，也可以抑制过拟合。

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。

一点补充 为了对齐Dropout训练和预测的结果，通常有两种做法，假设 $\text{dropout rate} = 0.2$ 。一种是训练时不作处理，预测时输出乘以 $(1 - \text{dropout rate})$ 。另一种是训练时留下的神经元除以 $(1 - \text{dropout rate})$ ，预测时不作处理。（[参考地址](#)）

5.5 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。虽然超参数的取值非常重要，但是在决定超参数的过程中一般会伴随很多的试错。

5.5.1 验证数据

这里要注意的是，不能使用测试数据评估超参数的性能。这一点非常重要，但也容易被忽视。

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。换句话说，用测试数据确认超参数的值的“好坏”，就会导致超参数的值被调整为只拟合测试数据。这样的话，可能就会得到不能拟合其他数据、泛化能力低的模型。

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。

训练数据用于参数（权重和偏置）的学习，验证数据用于超参数的性能评估。为了确认泛化能力，要在最后使用（比较理想的是只用一次）测试数据。

5.5.2 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。

所谓“大致地指定”，是指像 $0.001 (10^{-3})$ 到 $1000 (10^3)$ 这样，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”）。

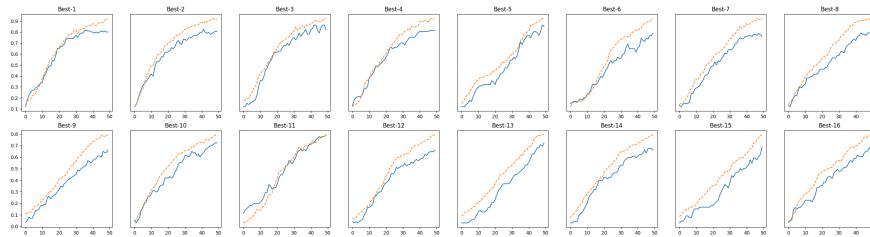


Figure 5.14: best 20 workouts

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。

- 步骤0：设定超参数的范围。
- 步骤1：从设定的超参数范围中随机采样。
- 步骤2：使用步骤1中采样到的超参数的值进行学习，通过验证数据评估识别精度（但是要将epoch设置得很小）。
- 步骤3：重复步骤1和步骤2（100次等），根据它们的识别精度的结果，缩小超参数的范围。

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。

5.5.3 超参数最优化的实现

Figure 5.14 展示最好的二十次训练，观察可以使学习顺利进行的超参数的范围，从而缩小值的范围。然后，在这个缩小的范围内重复相同的操作。

5.6 小结

- 参数的更新方法，除了 SGD 之外，还有 Momentum、AdaGrad、Adam 等方法。
- 权重初始值的赋值方法对进行正确的学习非常重要。
- 作为权重初始值，Xavier 初始值、He 初始值等比较有效。
- 通过使用 Batch Normalization，可以加速学习，并且对初始值变得健壮。
- 抑制过拟合的正则化技术有权值衰减、Dropout 等。
- 逐渐缩小“好值”存在的范围是搜索超参数的一个有效方法。

Chapter 6

卷积神经网络

本章的主题是卷积神经网络（Convolutional Neural Network, CNN）。CNN被用于图像识别、语音识别等各种场合。

6.1 整体结构

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）。

之前介绍的神经网络中，相邻层的所有神经元之间都有连接，这称为全连接（fully-connected）。另外，我们用Affine层实现了全连接层。如果使用这个Affine层，一个5层的全连接的神经网络就可以通过Figure 6.1所示的网络结构来实现。

如Figure 6.2所示CNN中新增了Convolution层和Pooling层。CNN的层的连接顺序是“Convolution - ReLU - (Pooling)”（Pooling层有时会被省略）。这可以理解为之前的“Affine - ReLU”连接被替换成了“Convolution - ReLU - (Pooling)”连接。

还需要注意的是，靠近输出的层中使用了之前的“Affine - ReLU”组合。此外，最后的输出层中使用了之前的“Affine - Softmax”组合。这些都是一般的CNN中比较常见的结构。

6.2 卷积层

CNN中出现了一些特有的术语，比如填充、步幅等。此外，各层中传递的数据是有形状的数据（比如，3维数据）。

6.2.1 全连接层存在的问题

之前介绍的全连接的神经网络中使用了全连接层（Affine层）。在全连接层中，相邻层的神经元全部连接在一起，输出的数量可以任意决定。

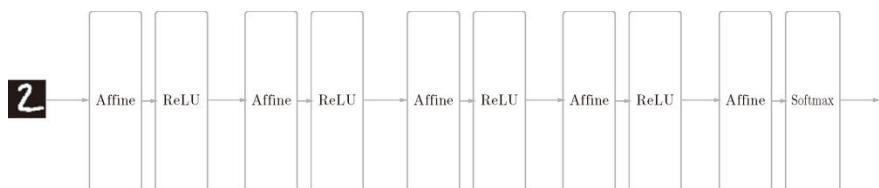


Figure 6.1: An example of a network based on a fully connected layer (Affine layer)

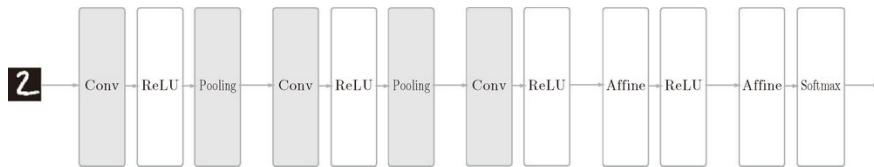


Figure 6.2: Examples of CNN-based networks

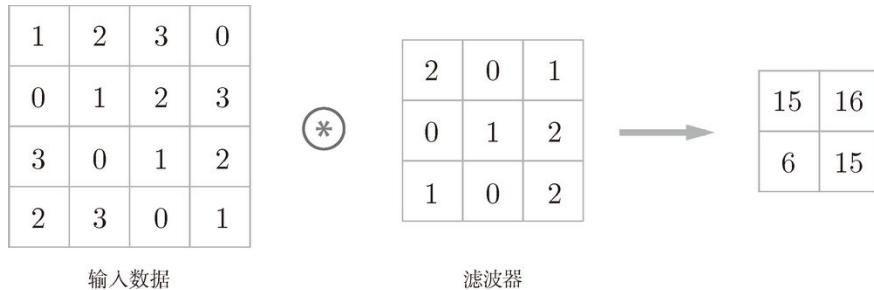


Figure 6.3: Example of convolution operation

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、RGB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。但是，因为全连接层会忽视形状，将全部的输入数据作为相同的神经元（同一维度的神经元）处理，所以无法利用与形状相关的信息。

而卷积层可以保持形状不变。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(feature map)。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)。

6.2.2 卷积运算

卷积层进行的处理就是卷积运算。卷积运算相当于图像处理中的“滤波器运算”(Figure 6.3)。

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。这里所说的窗口是指Figure 6.4中灰色的 3×3 的部分。如Figure 6.4所示，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。Figure 6.3的卷积运算的例子一直展示到了应用滤波器的阶段。包含偏置的卷积运算的处理流如图Figure 6.5所示。

如Figure 6.5所示，向应用了滤波器的数据加上了偏置。偏置通常只有1个(1×1)（本例中，相对于应用了滤波器的4个数据，偏置只有1个），这个值会被加到应用了滤波器的所有元素上。

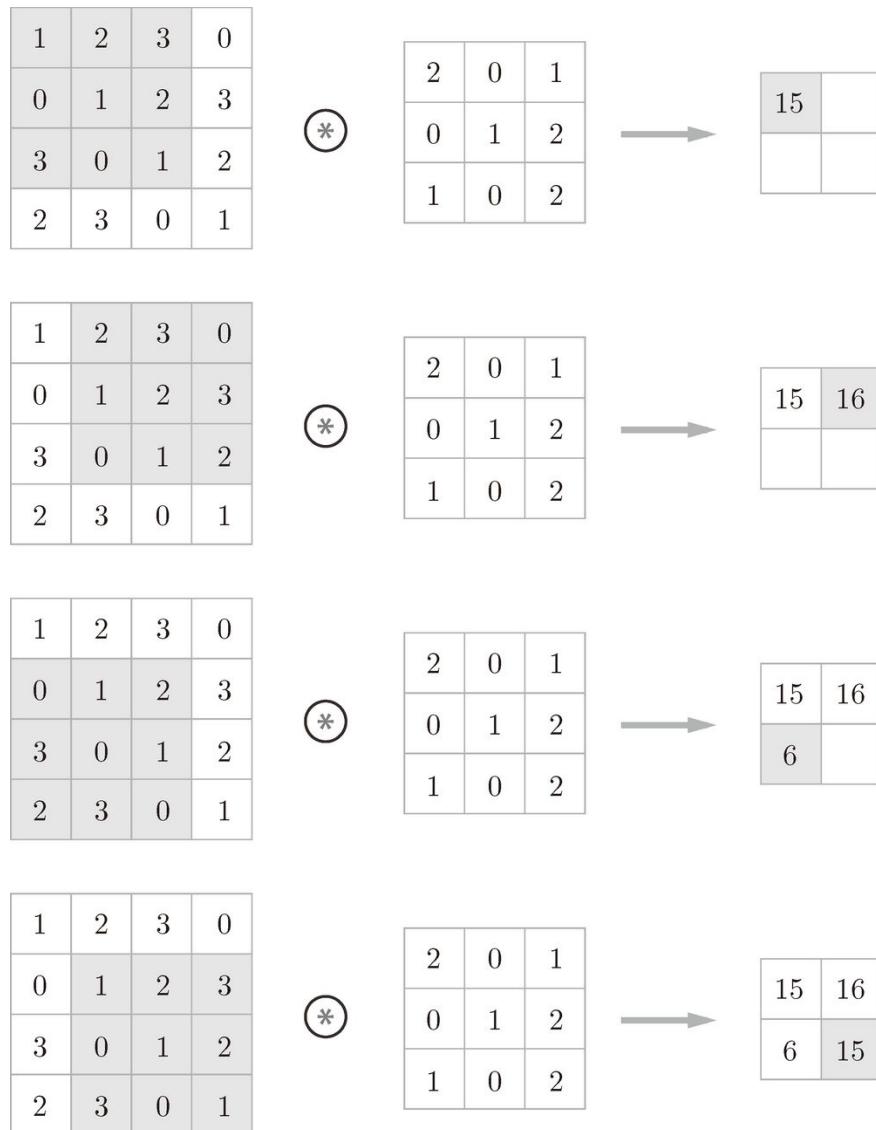


Figure 6.4: Calculation order of convolution operation

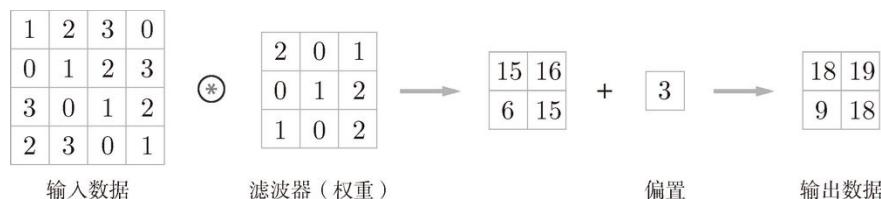


Figure 6.5: The bias of the convolution operation

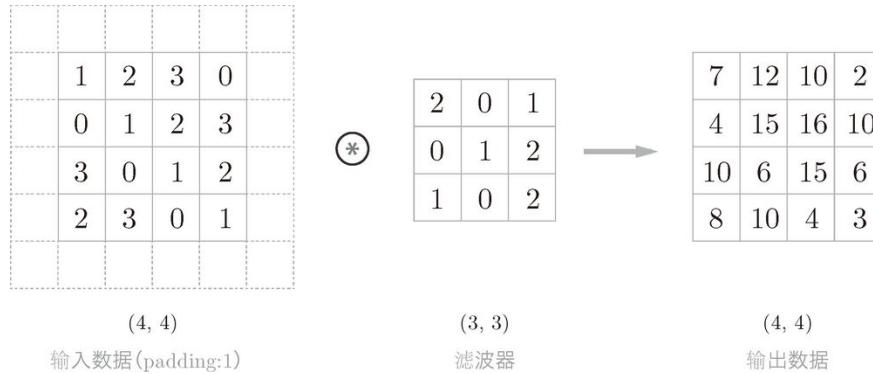


Figure 6.6: Filling processing of convolution operation

6.2.3 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），是卷积运算中经常会用到的处理。

如 Figure 6.6 所示，通过填充，大小为(4,4)的输入数据变成了(6,6)的形状。然后，应用大小为(3,3)的滤波器，生成了大小为(4,4)的输出数据。这个例子中将填充设成了1，不过填充的值也可以设置成2、3等任意的整数。

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。

6.2.4 步幅

应用滤波器的位置间隔称为步幅（stride）。在 Figure 6.7 的例子中，对输入大小为(7,7)的数据，以步幅2应用了滤波器。通过将步幅设为2，输出大小变为(3,3)。像这样，步幅可以指定应用滤波器的间隔。

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。

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 Equation 6.1 进行计算。

$$\begin{aligned} OH &= \frac{H + 2P - FH}{S} + 1 \\ OW &= \frac{W + 2P - FW}{S} + 1 \end{aligned} \tag{6.1}$$

这里需要注意的是，虽然只要代入值就可以计算输出大小，但是所设定的值必须使 Equation 6.1 中的和分别可以除尽。当输出大小无法除尽时（结果是小数时），需要采取报错等对策。顺便说一下，根据深度学习的框架的不同，当值无法除尽时，有时会向最接近的整数四舍五入，不进行报错而继续运行。

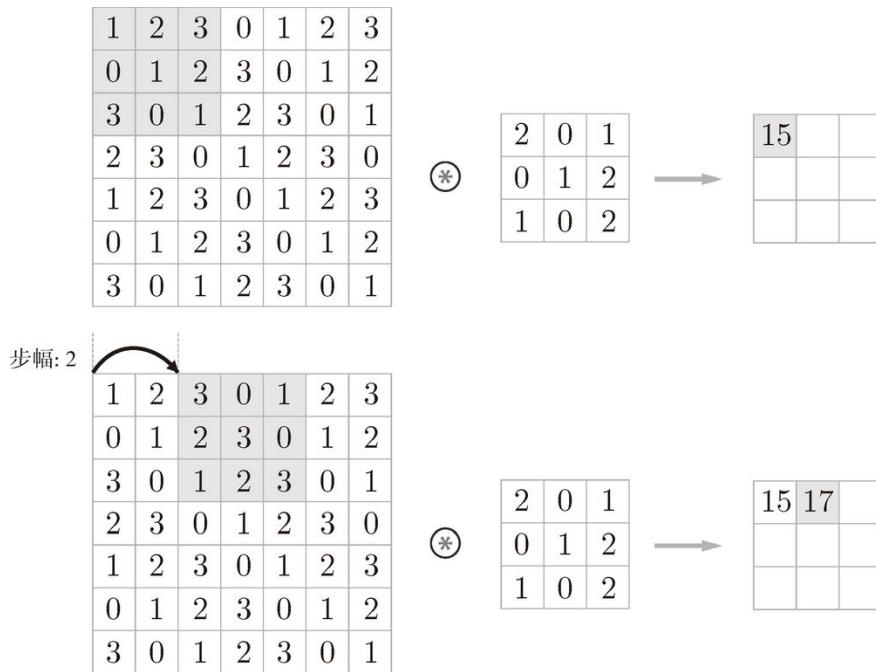


Figure 6.7: Example of convolution operation with stride 2

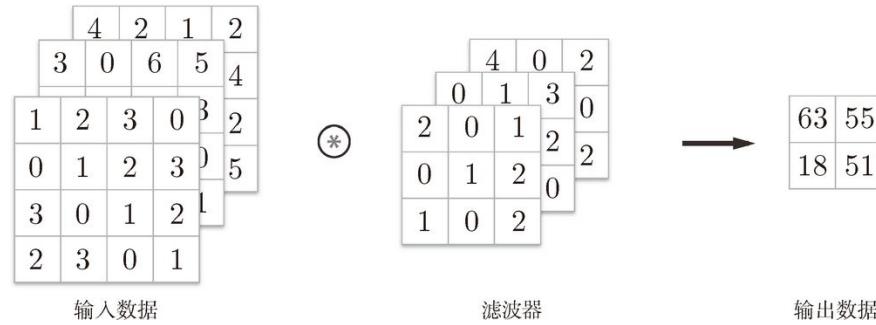


Figure 6.8: Example of convolution operation on 3D data

6.2.5 3维数据的卷积运算

之前的卷积运算的例子都是以有高、长方向的2维形状为对象的。但是，图像是3维数据，除了高、长方向之外，还需要处理通道方向。

Figure 6.8 是卷积运算的例子，**Figure 6.9** 是计算顺序。这里以3通道的数据为例，展示了卷积运算的结果。和2维数据时（图7-3的例子）相比，可以发现纵深方向（通道方向）上特征图增加了。通道方向上有多个特征图时，会按通道进行输入数据和滤波器的卷积运算，并将结果相加，从而得到输出。

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。滤波器大小可以设定为任意值（不过，每个通道的滤波器大小要全部相同）。这个例子中滤波器大小为(3,3)，但也可以设定为(2,2)、(1,1)、(5,5)等任意值。再强调一下，通道数只能设定为和输入数据的通道数相同的值。

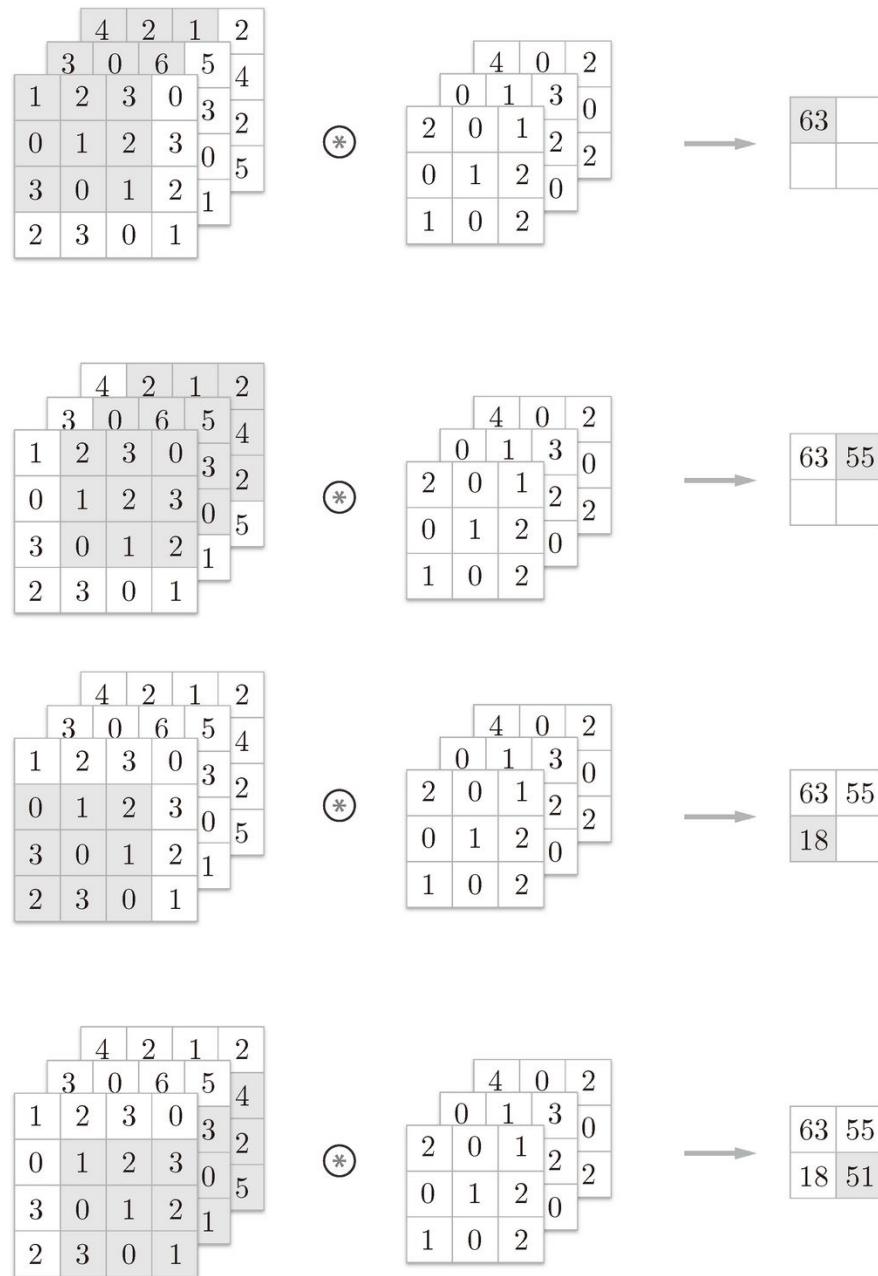


Figure 6.9: Computational order for convolution operations on 3D data

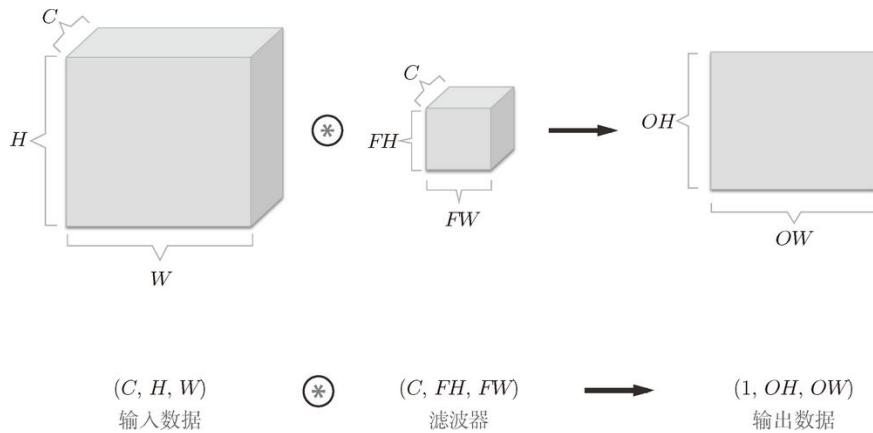


Figure 6.10: Thinking about convolution operations in combination with blocks

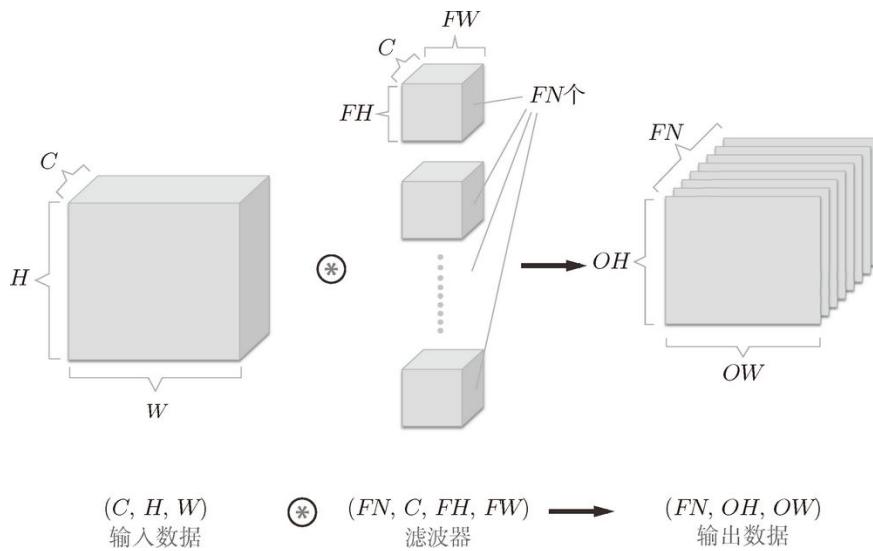


Figure 6.11: Example of convolution operation based on multiple filters

6.2.6 结合方块思考

将数据和滤波器结合长方体的方块来考虑，3维数据的卷积运算会很容易理解。方块是 Figure 6.10 所示的3维长方体。把3维数据表示为多维数组时，书写顺序为 (channel, height, width)。比如，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 (Filter Height)、长度为 FW (Filter Width) 时，可以写成 (C, FH, FW) 。

在 Figure 6.10 中，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。那么，如果要在通道方向上也拥有多个卷积运算的输出，就需要用到多个滤波器（权重）。用图表示的话，如 Figure 6.11。

Figure 6.11 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。

如 Figure 6.11 所示，关于卷积运算的滤波器，也必须考虑滤波器的数量。因此，作为4维数据，滤波器的权重数据要按 $(outputchannel, inputchannel, height, width)$ 的顺序书写。比如，通道数为 3、大小

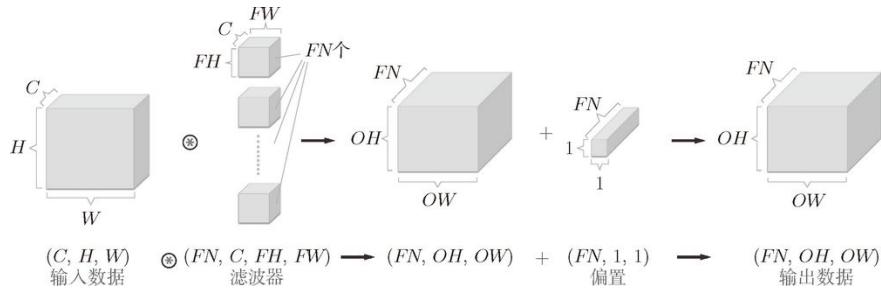


Figure 6.12: Processing flow of convolution operation

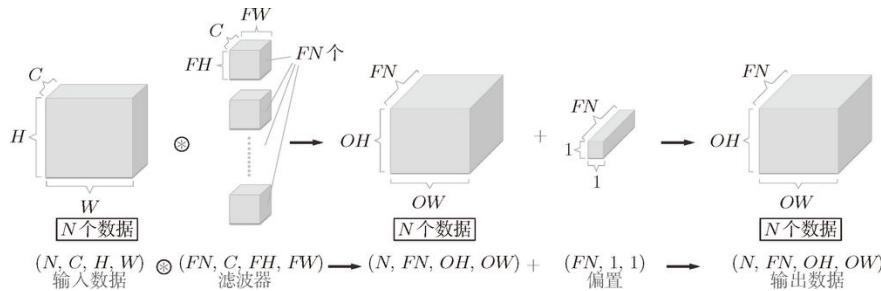


Figure 6.13: Processing flow of convolution operation (batch processing)

为 5×5 的滤波器有 20 个时，可以写成 $(20, 3, 5, 5)$ 。

卷积运算中（和全连接层一样）存在偏置。[Figure 6.12](#) 中，每个通道只有一个偏置。这里，偏置的形状是 $(FN, 1, 1)$ ，滤波器的输出结果的形状是 (FN, OH, OW) 。这两个方块相加时，要对滤波器的输出结果 (FN, OH, OW) 按通道加上相同的偏置值。

6.2.7 批处理

我们希望卷积运算也同样对应批处理。为此，需要将在各层间传递的数据保存为 4 维数据。具体地讲，就是按 $(batchnum, channel, height, width)$ 的顺序保存数据。比如，将 [Figure 6.12](#) 中的处理改成对 N 个数据进行批处理时，数据的形状如 [Figure 6.13](#) 所示。

[Figure 6.13](#) 的批处理版的数据流中，在各个数据的开头添加了批用的维度。像这样，数据作为 4 维的形状在各层间传递。这里需要注意的是，网络间传递的是 4 维数据，对这 N 个数据进行了卷积运算。也就是说，批处理将 N 次的处理汇总成了 1 次进行。

6.3 池化层

池化是缩小高、长方向上的空间的运算。

[Figure 6.14](#) 的例子是按步幅 2 进行 2×2 的 Max 池化时的处理顺序。“Max 池化”是获取最大值的运算，“ 2×2 ”表示目标区域的大小。如图所示，从 2×2 的区域中取出最大的元素。此外，这个例子中将步幅设为了 2，所以 2×2 的窗口的移动间隔为 2 个元素。另外，[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。

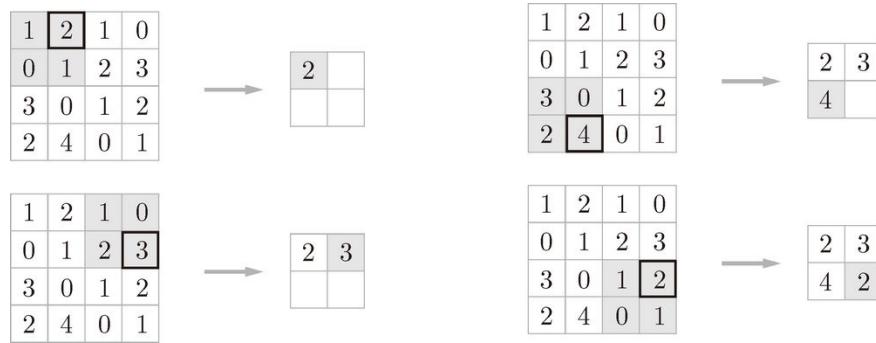


Figure 6.14: The processing order of Max pooling

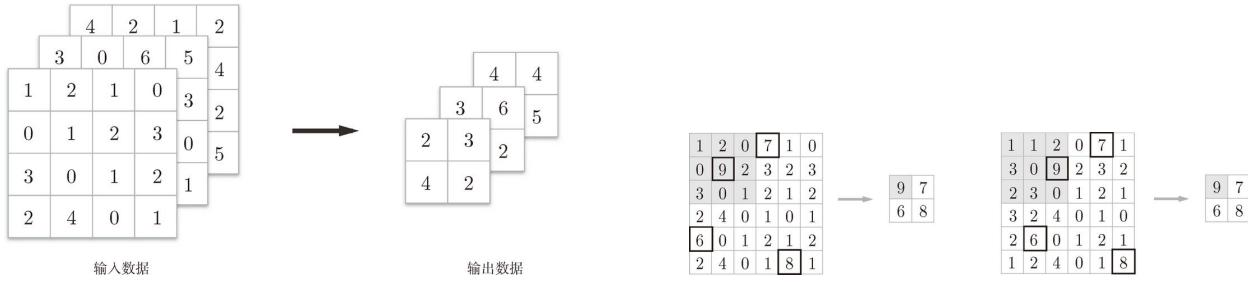


Figure 6.15: Features of the pooling layer

池化层的特征

没有要学习的参数 池化层和卷积层不同，没有要学习的参数。池化只是从目标区域中取最大值（或者平均值），所以不存在要学习的参数。

通道数不发生变化 经过池化运算，输入数据和输出数据的通道数不会发生变化。如 Figure 6.15a 所示，计算是按通道独立进行的。

对微小的位置变化具有鲁棒性（健壮） 输入数据发生微小偏差时，池化仍会返回相同的结果。因此，池化对输入数据的微小偏差具有鲁棒性(Figure 6.15b)。

6.4 卷积层和池化层的实现

6.4.1 4维数组

所谓4维数据，比如数据的形状是(10, 1, 28, 28)，则它对应10个高为28、长为28、通道为1的数据。

6.4.2 基于im2col的展开

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 Figure 6.16a 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）。

im2col会把输入数据展开以适合滤波器（权重）。具体地说，如 Figure 6.16b 所示，对于输入数据，将应用滤波器的区域（3维方块）横向展开为1列。im2col会在所有应用滤波器的地方进行这个展开处理。

在 Figure 6.16b 中，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

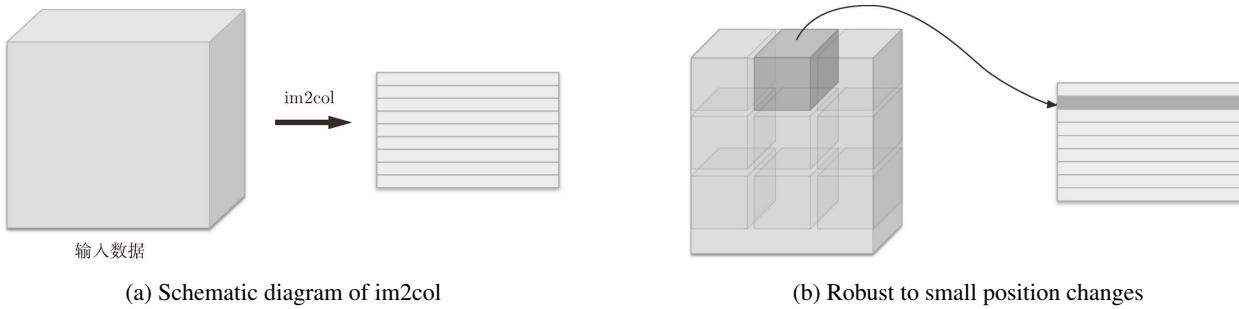


Figure 6.16: function im2col

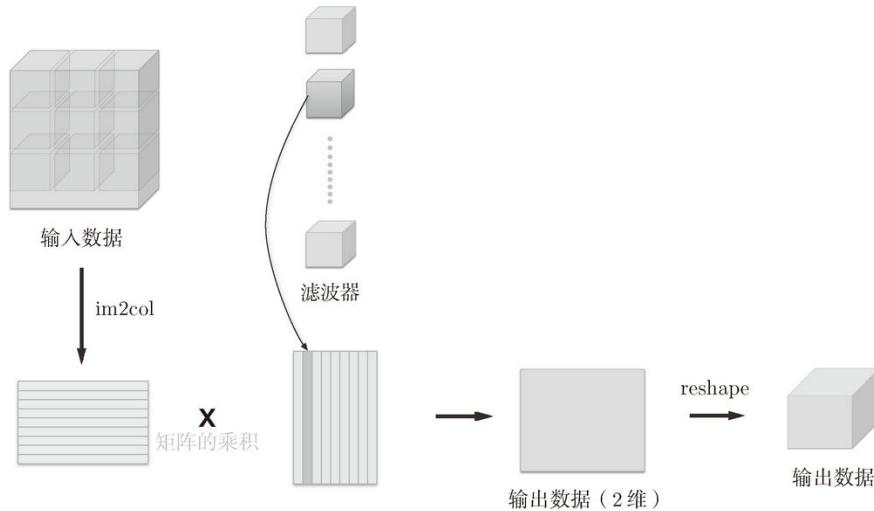


Figure 6.17: Details of filter processing for convolution operations

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。但是，汇总成一个大的矩阵进行计算，对计算机的计算颇有益处。比如，在矩阵计算的库（线性代数库）等中，矩阵计算的实现已被高度最优化，可以高速地进行大矩阵的乘法运算。因此，通过归结到矩阵计算上，可以有效地利用线性代数库。

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im2col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（参照Figure 6.17）。

如Figure 6.17所示，基于im2col方式的输出结果是2维矩阵。因为CNN中数据会保存为4维数组，所以要将2维输出数据转换为合适的形状。

6.4.3 卷积层的实现

im2col会将一个过滤器按照长、高、通道的顺序展开，然后进行步幅移动。

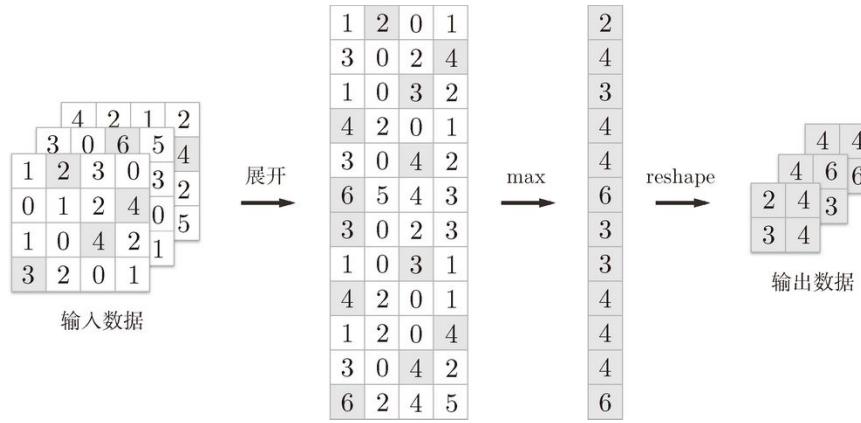


Figure 6.18: The implementation process of the pooling layer

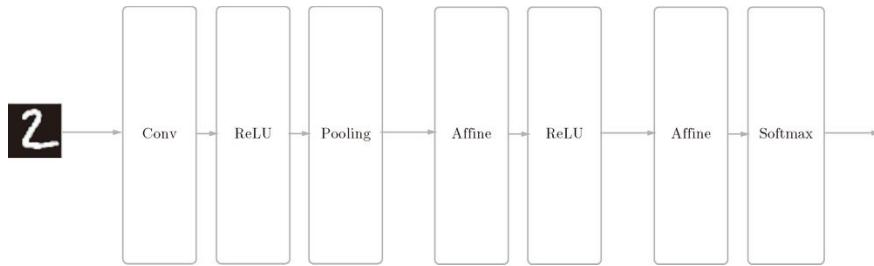


Figure 6.19: Network composition of a simple CNN

6.4.4 池化层的实现

池化层的实现和卷积层相同，都使用 im2col 展开输入数据。不过，池化的情况下，在通道方向上是独立的，这一点和卷积层不同。具体地讲，如 Figure 6.18 所示，池化的应用区域按通道单独展开。

像这样展开之后，只需对展开的矩阵求各行的最大值，并转换为合适的形状即可。

6.5 CNN的实现

如 Figure 6.19 所示，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。

6.6 CNN的可视化

6.6.1 第1层权重的可视化

6.6.2 基于分层结构的信息提取

6.7 具有代表性的CNN

关于CNN，迄今为止已经提出了各种网络结构。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。

6.7.1 LeNet

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。

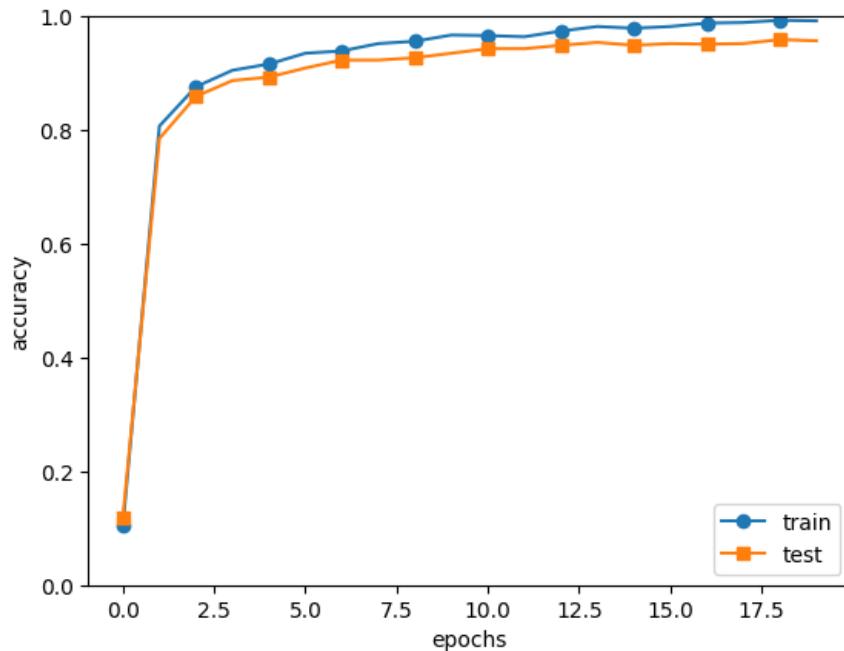


Figure 6.20: CNN training results on MNIST

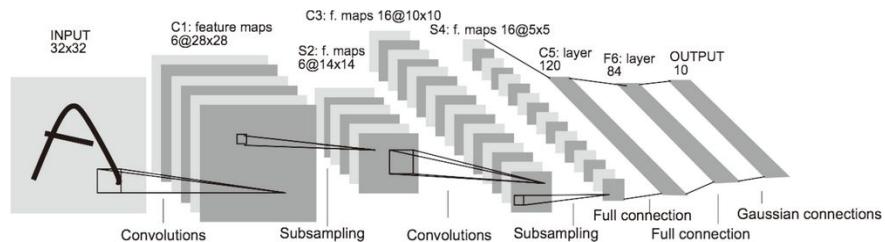


Figure 6.21: The network structure of LeNet

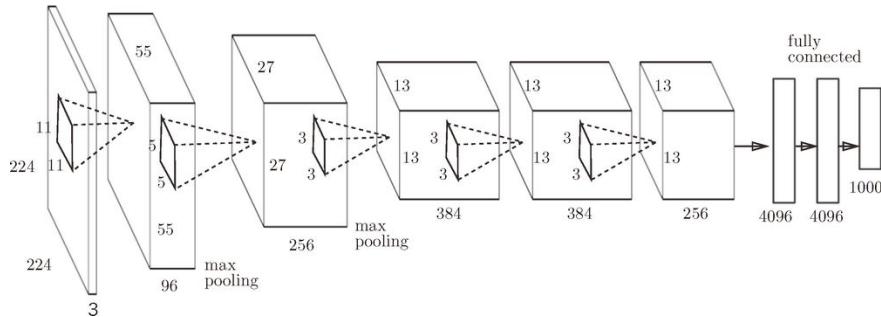


Figure 6.22: AlexNet

6.7.2 AlexNet

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。

6.8 小结

- CNN在此前的全连接层的网络中新增了卷积层和池化层。
- 使用im2col函数可以简单、高效地实现卷积层和池化层。

Chapter 7

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。

7.1 加深网络

7.1.1 向更深的网络出发

这里我们来创建一个如 Figure 7.1 所示的网络结构的CNN。

7.1.2 进一步提高识别精度

在一个标题为“**What is the class of this image ?**”的网站上，以排行榜的形式刊登了目前为止通过论文等渠道发表的针对各种数据集的方法的识别精度。排行榜中前几名的方法，可以发现进一步提高识别精度的技术和线索。比如，集成学习、学习率衰减、**Data Augmentation**（数据扩充）等都有助于提高识别精度。尤其是Data Augmentation，虽然方法很简单，但在提高识别精度上效果显著。

Data Augmentation基于算法“人为地”扩充输入图像（训练图像）。具体地说，如Figure 7.3所示，对于输入图像，通过施加旋转、垂直或水平方向上的移动等微小变化，增加图像的数量。这在数据集的图像数量有限时尤其有效。

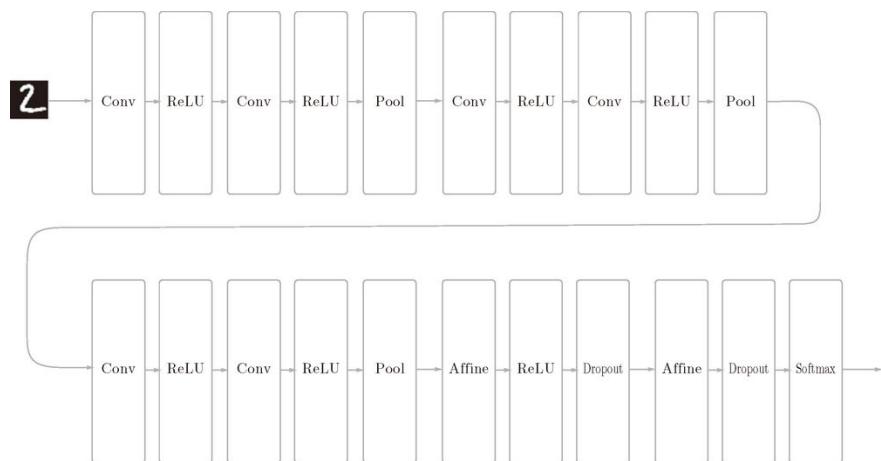


Figure 7.1: Deep CNN for Handwritten Digit Recognition

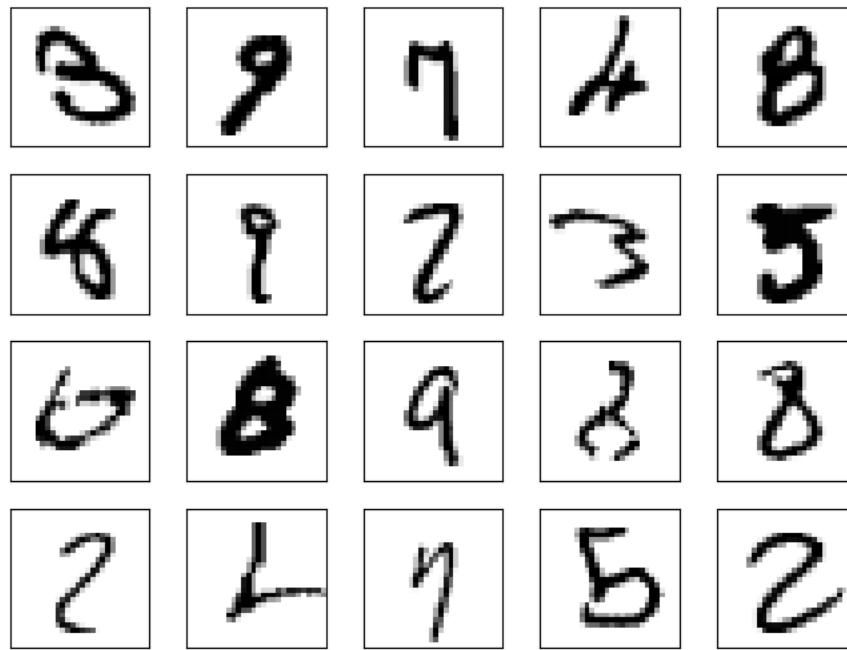


Figure 7.2: Examples of misidentified images

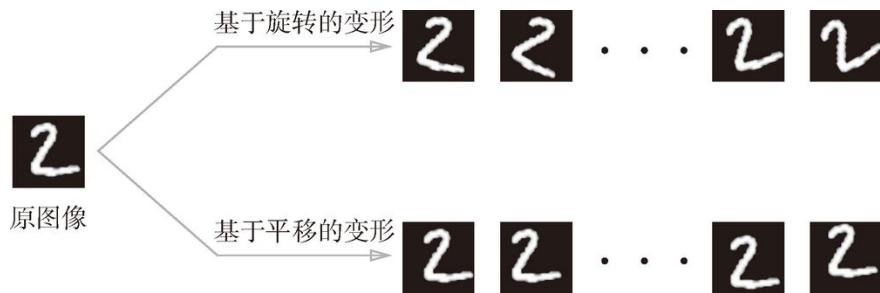


Figure 7.3: Data Augmentation example

除了如 Figure 7.3 所示的变形之外，Data Augmentation还可以通过其他各种方法扩充图像，比如裁剪图像的“crop处理”、将图像左右翻转的“flip处理”¹等。对于一般的图像，施加亮度等外观上的变化、放大缩小等尺度上的变化也是有效的。不管怎样，通过Data Augmentation巧妙地增加训练图像，就可以提高深度学习的识别精度。虽然这个看上去只是一个简单的技巧，不过经常会有很好的效果。

7.1.3 加深层的动机

关于加深层的重要性，现状是理论研究还不够透彻。尽管目前相关理论还比较贫乏，但是有几点可以从过往的研究和实验中得以解释。

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。

¹ flip处理只在不需要考虑图像对称性的情况下有效。

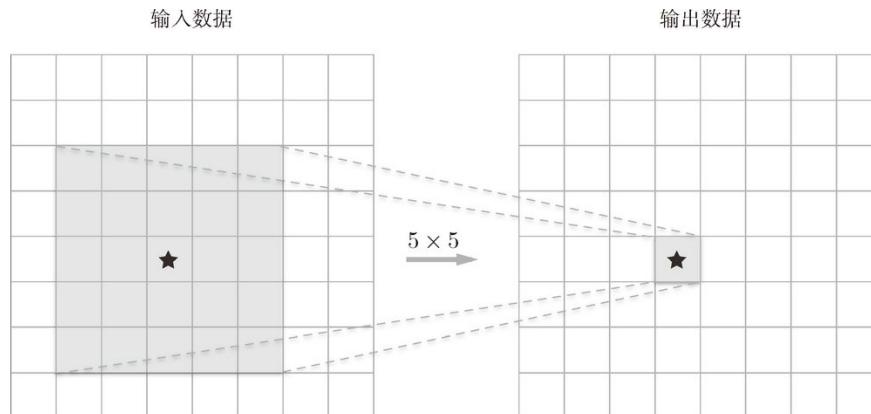


Figure 7.4: 5-5 convolution example

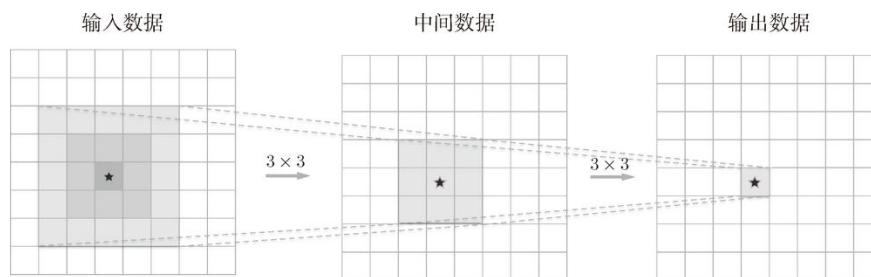


Figure 7.5: Example of a convolutional layer repeated twice 3-3

显然，在 Figure 7.4 的例子中，每个输出节点都是从输入数据的某个 5×5 的区域算出来的。接下来我们思考一下 Figure 7.5 中重复两次 5×5 的卷积运算的情形。此时，每个输出节点将由中间数据的某个 3×3 的区域计算出来。

一次 5×5 的卷积运算的区域可以由两次 3×3 的卷积运算抵充。并且，相对于前者的参数数量 25 (5×5)，后者一共是 18 ($2 \times 3 \times 3$)，通过叠加卷积层，参数数量减少了。而且，这个参数数量之差会随着层的加深而变大。比如，重复三次 3×3 的卷积运算时，参数的数量总共是 27。而为了用一次卷积运算“观察”与之相同的区域，需要一个 7×7 的滤波器，此时的参数数量是 49。

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。

7.2 深度学习的小历史

7.2.1 VGG

VGG 是由卷积层和池化层构成的基础的 CNN。不过，如图 8-9 所示，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”）

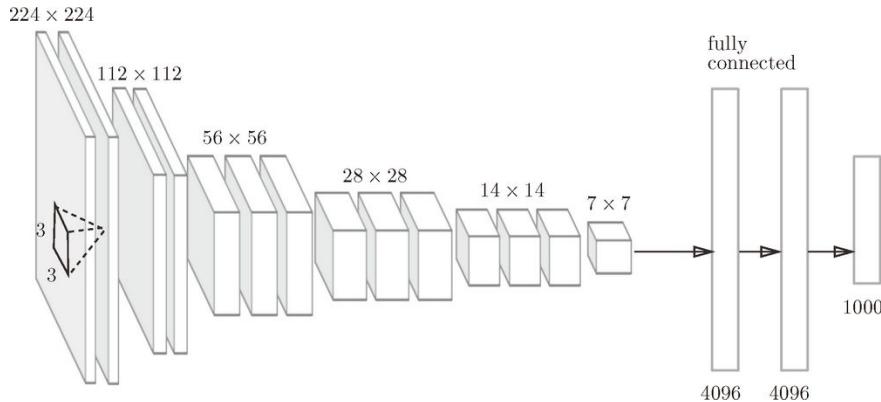


Figure 7.6: VGG

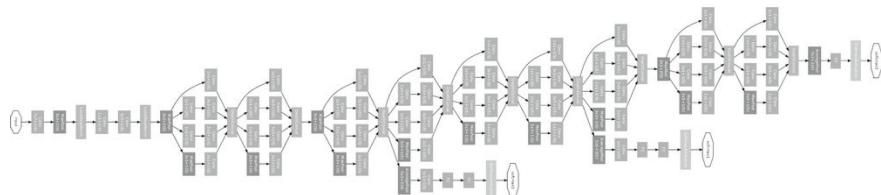


Figure 7.7: GoogLeNet

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Figure 7.6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。

7.2.2 GoogLeNet

只看Figure 7.7的话，这似乎是一个看上去非常复杂的网络结构，但实际上它基本上和之前介绍的CNN结构相同。不过，GoogLeNet的特征是，网络不仅在纵向上有深度，在横向上也有深度（广度）。

GoogLeNet在横向上有“宽度”，这称为“Inception结构”。

Figure 7.8, 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。

7.2.3 ResNet

ResNet是微软团队开发的网络。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）。

如Figure 7.9所示，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。Figure 7.9中，在连续2层的卷积层中，将输入 x 跳着连接至2层后的输出。这里的重点是，通过快捷结构，原

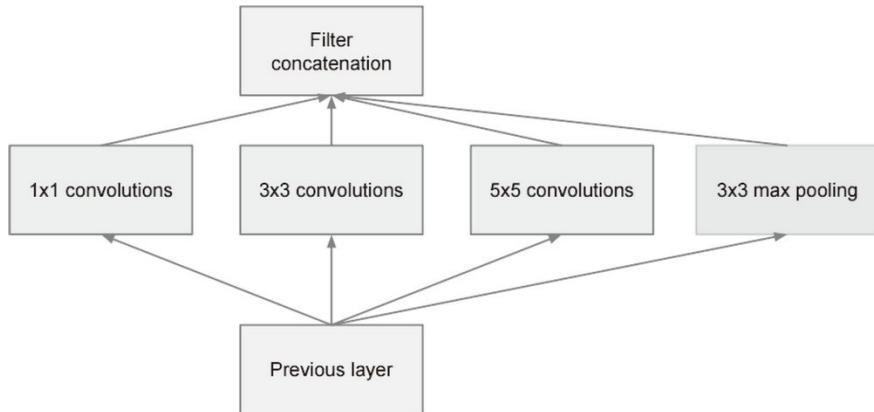


Figure 7.8: Inception structure of GoogLeNet

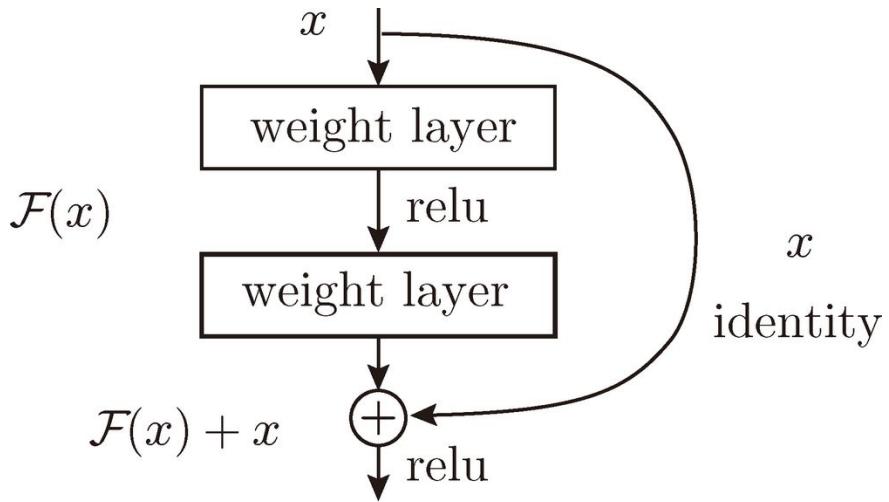


Figure 7.9: Components of ResNet

来的2层卷积层的输出 $\mathcal{F}(x)$ 变成了 $\mathcal{F}(x) + x$ 。通过引入这种快捷结构，即使加深层，也能高效地学习。这是因为，通过快捷结构，反向传播时信号可以无衰减地传递。

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消失问题就有望得到缓解。

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。

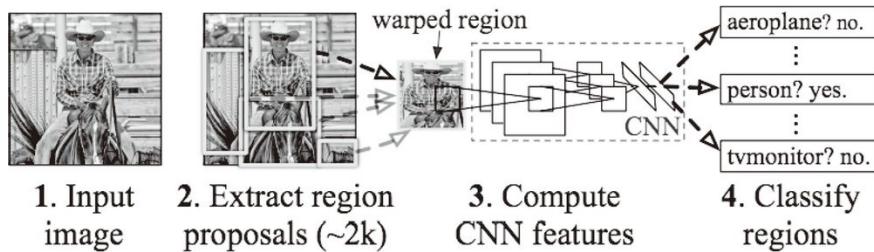


Figure 7.10: Processing flow of R-CNN

7.3 深度学习的高速化

7.3.1 需要努力解决的问题

7.3.2 基于GPU的高速化

GPU计算，是指基于GPU进行通用的数值计算的操作。

深度学习中需要进行大量的乘积累加运算（或者大型矩阵的乘积运算）。这种大量的并行运算正是GPU所擅长的（反过来说，CPU比较擅长连续的、复杂的计算）。

7.3.3 分布式学习

7.3.4 运算精度的位数缩减

7.4 深度学习的应用案例

7.4.1 物体检测

物体检测是从图像中确定物体的位置，并进行分类的问题。

对于这样的物体检测问题，人们提出了多个基于CNN的方法。这些方法展示了非常优异的性能，并且证明了在物体检测的问题上，深度学习是非常有效的。

在使用CNN进行物体检测的方法中，有一个叫作R-CNN的有名的方法。[Figure 7.10](#) 显示了R-CNN的处理流。

7.4.2 图像分割

图像分割是指在像素水平上对图像进行分类。使用以像素为单位对各个对象分别着色的监督数据进行学习。然后，在推理时，对输入图像的所有像素进行分类。

要基于神经网络进行图像分割，最简单的方法是以所有像素为对象，对每个像素执行推理处理。比如，准备一个对某个矩形区域中心的像素进行分类的网络，以所有像素为对象执行推理处理。正如大家能想到的，这样的方法需要按照像素数量进行相应次 forward 处理，因而需要耗费大量的时间（正确地说，卷积运算中会发生重复计算很多区域的无意义的计算）。为了解决这个无意义的计算问题，有人提出了一个名为FCN（Fully Convolutional Network）的方法。该方法通过一次 forward 处理，对所有像素进行分类。

FCN的字面意思是“全部由卷积层构成的网络”。相对于一般的CNN包含全连接层，FCN将全连接层替换成为发挥相同作用的卷积层。

全连接层中，输出和全部的输入相连。使用卷积层也可以实现与此结构完全相同的连接。比如，针对输入大小是 $32 \times 10 \times 10$ （通道数32、高10、长10）的数据的全连接层可以替换成滤波器

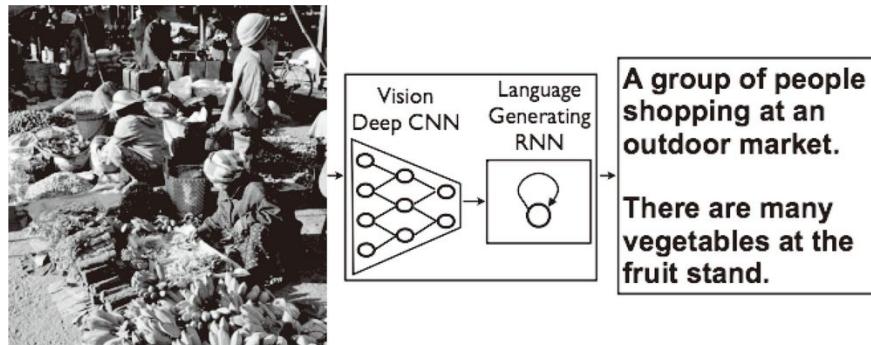


Figure 7.11: Neural Image Caption

大小为 $32 \times 10 \times 10$ 的卷积层。如果全连接层的输出节点数是 100，那么在卷积层准备 100 个 $32 \times 10 \times 10$ 的滤波器就可以实现完全相同的处理。像这样，全连接层可以替换成进行相同处理的卷积层。

7.4.3 图像标题生成

一个基于深度学习生成图像标题的代表性方法是被称为 NIC (Neural Image Caption) 的模型。如 NIC Figure 7.11 所示，NIC 由深层的 CNN 和处理自然语言的 RNN (Recurrent Neural Network) 构成。RNN 是呈递归式连接的网络，经常被用于自然语言、时间序列数据等连续性的数据上。

NIC 基于 CNN 从图像中提取特征，并将这个特征传给 RNN。RNN 以 CNN 提取出的特征为初始值，递归地生成文本。基本上 NIC 是组合了两个神经网络 (CNN 和 RNN) 的简单结构。基于 NIC，可以生成惊人的高精度的图像标题。我们将组合图像和自然语言等多种信息进行的处理称为多模态处理。

多模态处理

RNN 的 R 表示 Recurrent (递归的)。这个递归指的是神经网络的递归的网络结构。根据这个递归结构，神经网络会受到之前生成的信息的影响（换句话说，会记忆过去的信息），这是 RNN 的特征。比如，生成“我”这个词之后，下一个要生成的词受到“我”这个词的影响，生成了“要”；然后，再受到前面生成的“我要”的影响，生成了“睡觉”这个词。对于自然语言、时间序列数据等连续性的数据，RNN 以记忆过去的信息的方式运行。

7.5 深度学习的未来

7.5.1 图像风格变换

输入两个图像后，会生成一个新的图像。两个输入图像中，一个称为“内容图像”，另一个称为“风格图像”。此项研究出自论文 “A Neural Algorithm of Artistic Style”，在学习过程中使网络的中间数据近似内容图像的中间数据。这样一来，就可以使输入图像近似内容图像的形状。此外，为了从风格图像中吸收风格，导入了风格矩阵的概念。通过在学习过程中减小风格矩阵的偏差。

7.5.2 图像的生成

能画出以假乱真的图像的 DCGAN 会将图像的生成过程模型化。使用大量图像（比如，印有卧室的 DCGAN 大量图像）训练这个模型，学习结束后，使用这个模型，就可以生成新的图像。

DCGAN 中使用了深度学习，其技术要点是使用了 Generator (生成者) 和 Discriminator (识别者) 这两个神经网络。Generator 生成近似真品的图像，Discriminator 判别它是不是真图像（是 Generator 生

成的图像还是实际拍摄的图像)。像这样,通过让两者以竞争的方式学习,Generator会学习到更加精妙的图像作假技术,Discriminator则会成长为能以更高精度辨别真假的鉴定师。两者互相切磋、共同成长,这是GAN(Generative Adversarial Network)这个技术的有趣之处。在这样的切磋中成长起来的Generator最终会掌握画出足以以假乱真的图像的能力(或者说有这样的可能)。

7.5.3 自动驾驶

如果可以在各种环境中稳健地正确识别行驶区域的话,实现自动驾驶可能也就没那么遥远了。最近,在识别周围环境的技术中,深度学习的力量备受期待。比如,基于CNN的神经网络SegNet。

7.5.4 Deep Q-Network(强化学习)

就像人类通过摸索试验来学习一样(比如骑自行车),让计算机也在摸索试验的过程中自主学习,这称为强化学习(reinforcement learning)。

强化学习的基本框架是,代理(Agent)根据环境选择行动,然后通过这个行动改变环境。根据环境的变化,代理获得某种报酬。强化学习的目的是决定代理的行动方针,以获得更好的报酬。这里需要注意的是,报酬并不是确定的,只是“预期报酬”。

在使用了深度学习的强化学习方法中,有一个叫作Deep Q-Network(通称DQN)的方法。该方法基于被称为Q学习的强化学习算法。在Q学习中,为了确定最合适行动,需要确定一个被称为最优行动价值函数的函数。为了近似这个函数,DQN使用了深度学习(CNN)。

Appendix A

Softmax-with-Loss层的计算图

A.1 反向传播

A.1.1 Cross Entropy Error层

A.1.2 Softmax层

正向传播时若有分支流出，则反向传播时它们的反向传播的值会相加。

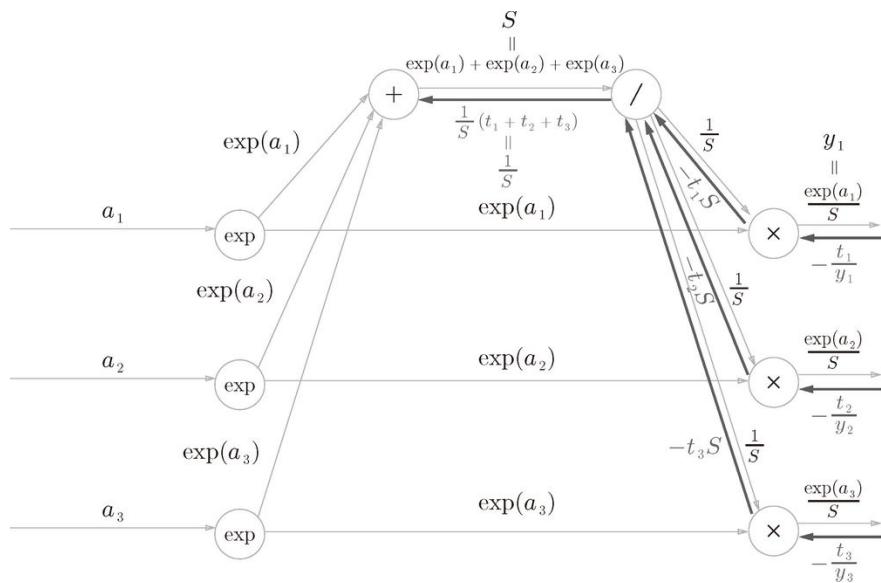


Figure A.1: Step 3