深度学习入门

基于Python的理论与实现

Stephen CUI

March 14, 2023

Chapter 1

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。

1.1 从感知机到神经网络

1.1.1 神经网络的例子

用图来表示神经网络的话，如Figure 1.1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。

1.1.2 复习感知机

引入新函数$h(x)$，将**??**改写成下面的方程：

$$y = h(b + w_1 x_1 + w_2 x_2) \tag{1.1}$$

$$h(x) = \begin{cases} 0 & ,x \leq 0 \\ 1 & ,x > 0 \end{cases} \tag{1.2}$$

1.1.3 激活函数登场

Equation 1.2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。

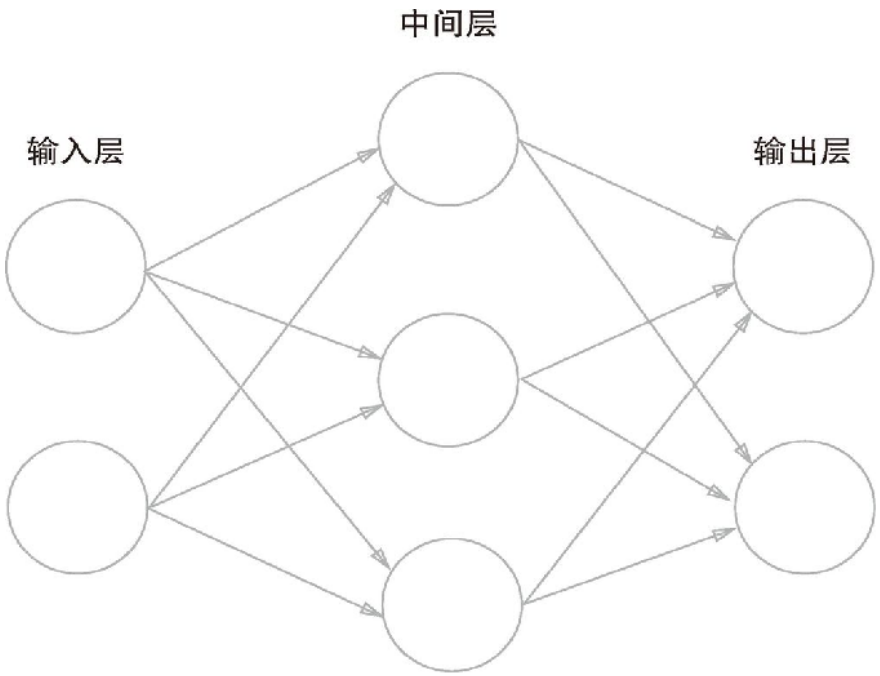

Figure 1.1: Examples of Neural Networks

1.2 激活函数

Equation 1.2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。

1.2.1 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）：

$$h(x) = \frac{1}{1+\exp(-x)} \tag{1.3}$$

1.2.2 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

1.2.3 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。

1.2.4 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:

$$h(x) = \begin{cases} x & ,x > 0 \\ 0 & ,x \leq 0 \end{cases} \tag{1.4}$$

1.3 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。

1.4 3层神经网络的实现

图3-17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]。

1.5 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。

1.5.1 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：

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)} \tag{1.5}$$

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]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。

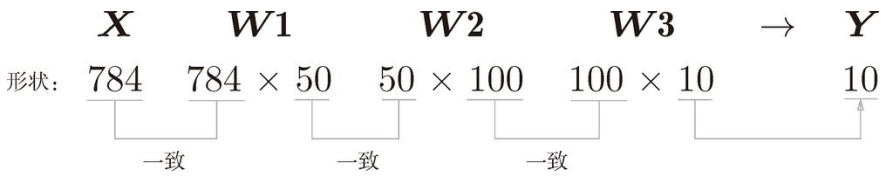

Figure 1.2: change of array shape

1.5.2 实现softmax函数时的注意事项

Equation 1.5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。

$$\begin{aligned} y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n}\exp(a_i)} &= \frac{C\exp(a_k)}{C\sum_{i=1}^{n}\exp(a_i)} \\ &= \frac{\exp(a_k+\log C)}{\sum_{i=1}^{n}\exp(a_i+\log C)} \\ &= \frac{\exp(a_k+C')}{\sum_{i=1}^{n}\exp(a_i+C')} \end{aligned} \tag{1.6}$$

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。

1.5.3 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。

1.5.4 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。

1.6 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）。

1.6.1 批处理

从整体的处理流程来看，Figure 1.2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。

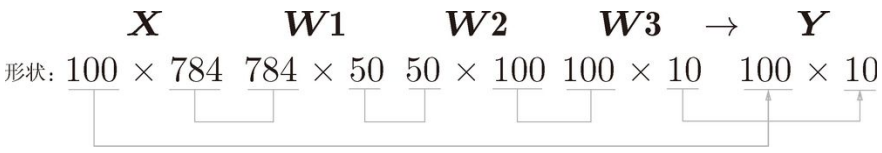

Figure 1.3: Variation of array shape in batch

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。

1.7 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 2

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。

2.1 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 2.1中把“×2”“×1.1”等作为一个运算整体用〇括起来了，不过只用◯表示乘法运算“×”也是可行的。此时，如Figure 2.2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。

2.1.1 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。

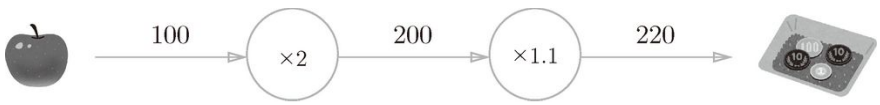

Figure 2.1: Based on the calculation graph to solve the answer

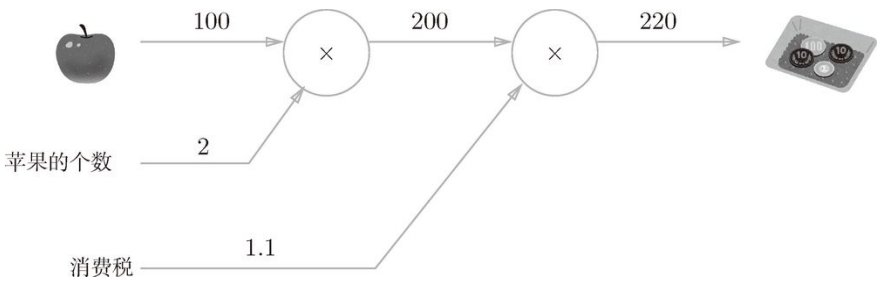

Figure 2.2: Based on the calculation graph to solve the answer2

2.2 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。

2.2.1 计算图的反向传播

2.2.2 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：

$$\begin{aligned} z &= t^2 \\ t &= x+y \end{aligned} \tag{2.1}$$

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 2.1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：

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x} \tag{2.2}$$

对于$z=(x+y)^2$，那么就有

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

2.2.3 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$\frac{\partial z}{\partial z}$，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。

2.3 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“$\times$”等运算为例，介绍反向传播的结构。

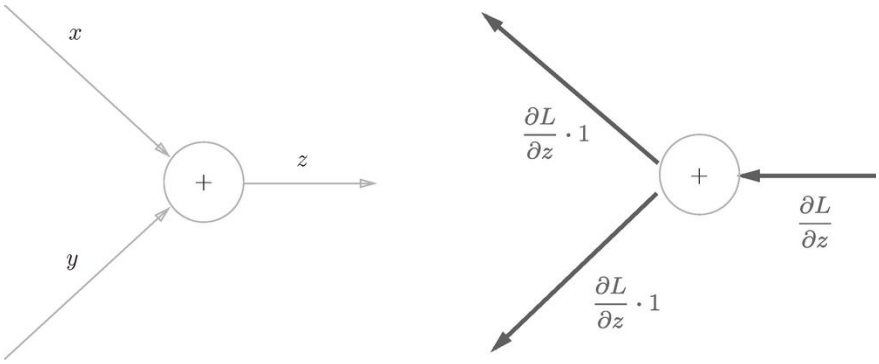

Figure 2.3: Backpropagation for Adder nodes

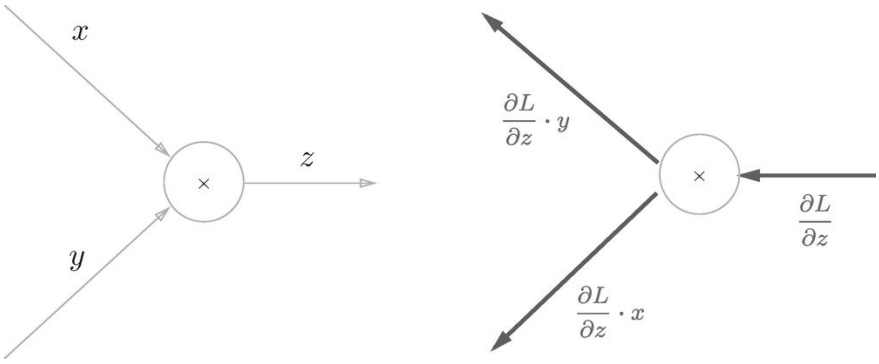

Figure 2.4: Backpropagation of multiply nodes

2.3.1 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：

$$\frac{\partial z}{\partial x} = 1$$
$$\frac{\partial z}{\partial y} = 1$$

Figure 2.3, 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。

2.3.2 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：

$$\frac{\partial z}{\partial x} = y$$
$$\frac{\partial z}{\partial y} = x$$

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。

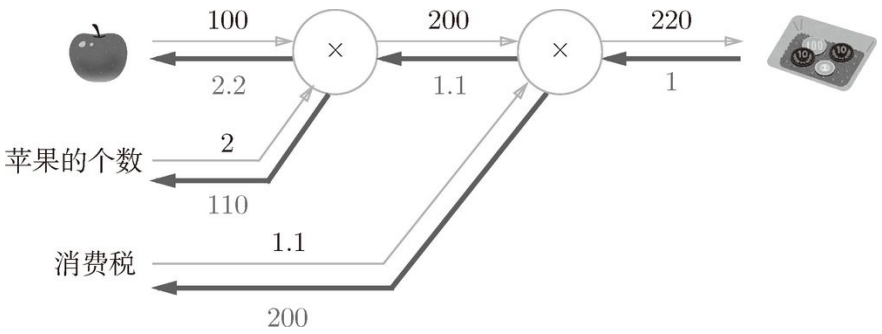

Figure 2.5: Example of backpropagation for buying apples

2.4 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）。

2.4.1 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。

2.5 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。

2.5.1 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：

$$\frac{\partial y}{\partial x} = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases} \tag{2.3}$$

Equation 2.3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。

2.5.2 Sigmoid层

sigmoid函数由下式表示：

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

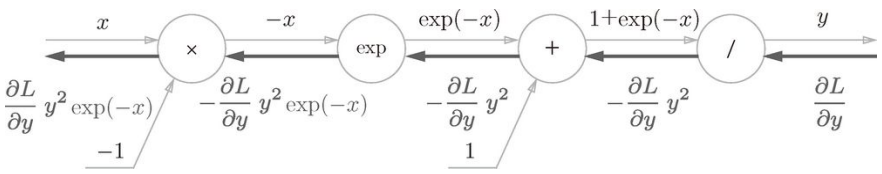

Figure 2.6: Computational graph of the Sigmoid layer

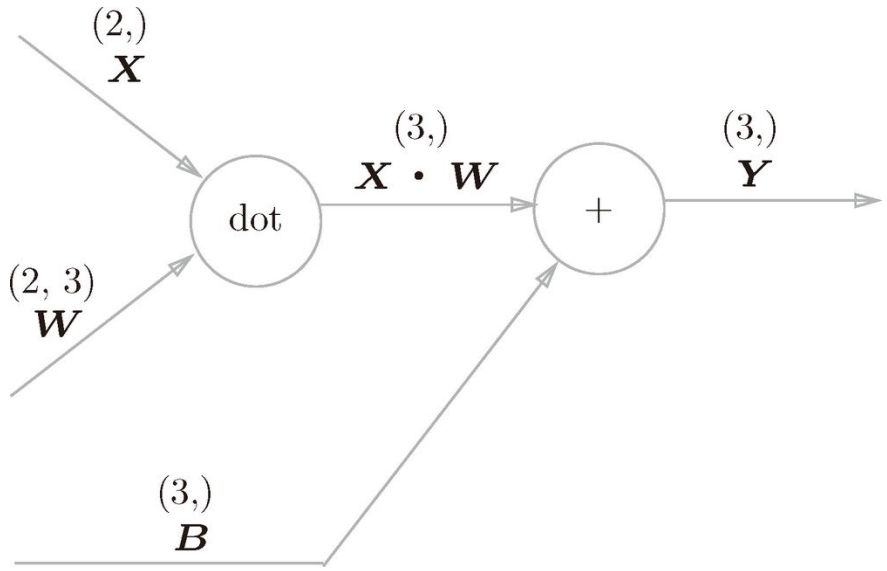

Figure 2.7: Computational graph of the Affine layer

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

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

2.6 Affine/Softmax层的实现

2.6.1 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 2.7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）

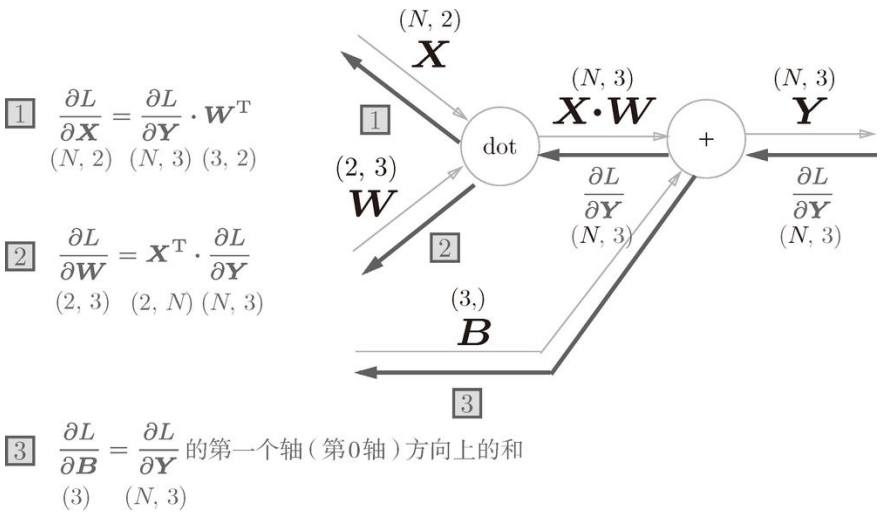

Figure 2.8: Computational graph of the batch version of the Affine layer

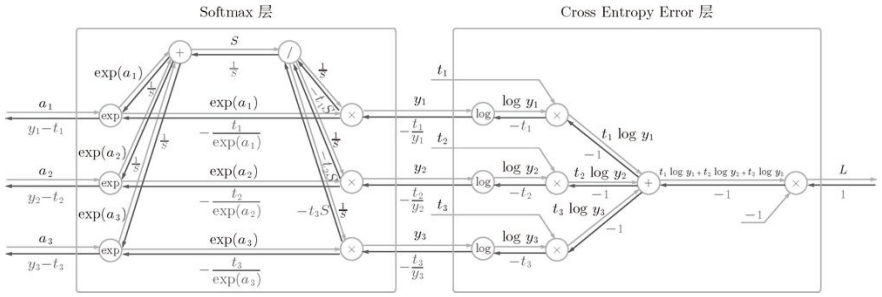

Figure 2.9: Computational graph of the Softmax-with-Loss layer

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

$$\begin{aligned}\frac{\partial L}{\partial \boldsymbol{X}} &= \frac{\partial L}{\partial \boldsymbol{Y}}\boldsymbol{W}^T \\ \frac{\partial L}{\partial \boldsymbol{W}} &= \boldsymbol{X}^T\frac{\partial L}{\partial \boldsymbol{Y}}\end{aligned} \tag{2.4}$$

2.6.2 批版本的Affine层

前面介绍的Affine层的输入$\boldsymbol{X}$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。

2.6.3 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 2.9所示。

反向传播的具体的推导参见Appendix A

Figure 2.10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 2.10 所示，Softmax 层将输入$(a1, a2, a3)$正

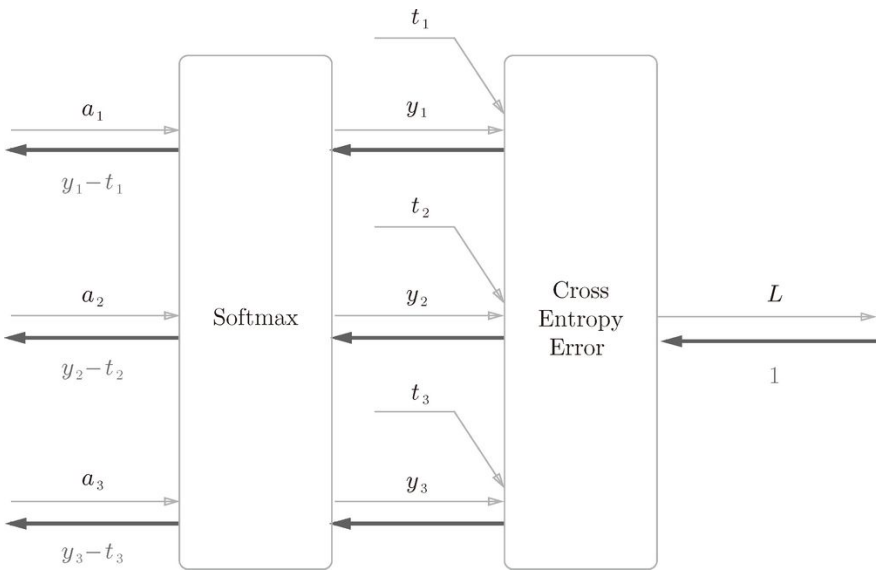

Figure 2.10: Simple version of the calculation graph of the Softmax-with-Loss layer

规化，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。

2.7 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。

2.7.1 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）。

2.7.2 使用误差反向传播法的学习

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。

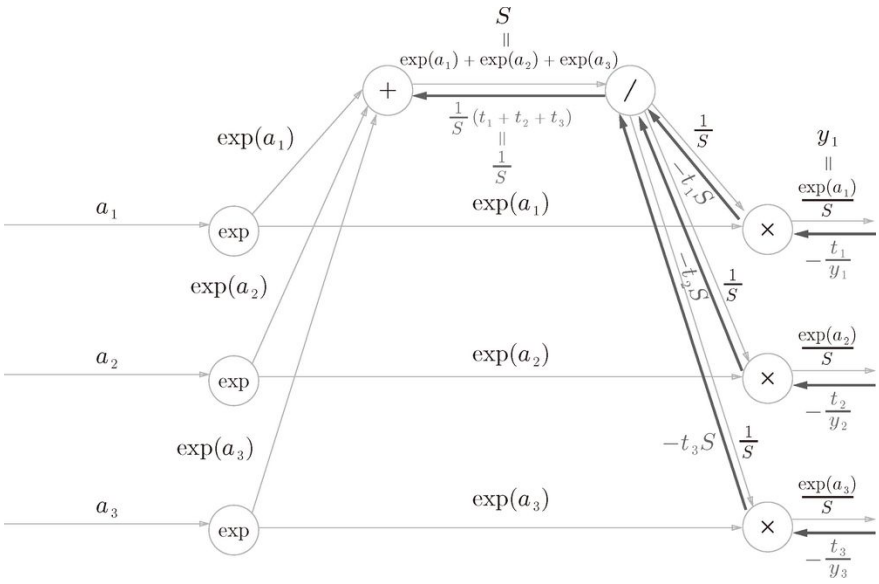

Figure A.1: Step 3