



# 深度学习：零基础入门

从零实现深度学习 (Python 版)

作者：Bingtao Han

组织：Open Access

时间：October 30, 2019

版本：0.09



*Victory won't come to us unless we go to it. — M. Moore*

# 目 录

---

<b>1 感知器</b>	<b>2</b>	4.7 卷积神经网络的应用 . . . . .	<b>74</b>
1.1 深度学习是啥 . . . . .	2	4.8 小结 . . . . .	<b>75</b>
1.2 感知器 . . . . .	3	<b>5 循环神经网络</b>	<b>76</b>
1.3 编程实战：实现感知器 . . .	5	5.1 语言模型 . . . . .	<b>76</b>
1.4 小结 . . . . .	8	5.2 循环神经网络是啥 . . . . .	<b>77</b>
<b>2 线性单元和梯度下降</b>	<b>10</b>	5.3 循环神经网络的训练 . . . . .	<b>80</b>
2.1 线性单元是啥 . . . . .	10	5.4 RNN 的应用：基于 RNN 的	
2.2 编程实战：实现线性单元 . .	16	语言模型 . . . . .	<b>87</b>
2.3 小结 . . . . .	18	5.5 编程实战：RNN 的实现 . . .	<b>91</b>
<b>3 神经网络和反向传播算法</b>	<b>20</b>	5.6 小节 . . . . .	<b>94</b>
3.1 神经元 . . . . .	20	<b>6 长短时记忆网络</b>	<b>95</b>
3.2 神经网络是啥 . . . . .	21	6.1 长短时记忆网络是啥 . . . . .	<b>95</b>
3.3 计算神经网络的输出 . . . . .	22	6.2 长短时记忆网络的前向计算 .	<b>97</b>
3.4 神经网络的矩阵表示 . . . . .	23	6.3 长短时记忆网络的训练 . . . .	<b>100</b>
3.5 神经网络的训练 . . . . .	25	6.4 编程实战：长短时记忆网络	
3.6 编程实战：神经网络的实现	28	的实现 . . . . .	<b>106</b>
3.7 神经网络实战：手写数字识别	37	6.5 GRU . . . . .	<b>114</b>
3.8 小结 . . . . .	45	6.6 小结 . . . . .	<b>115</b>
<b>4 卷积神经网络</b>	<b>47</b>	<b>7 循环神经网络</b>	<b>116</b>
4.1 激活函数:Relu . . . . .	47	7.1 递归神经网络是啥 . . . . .	<b>116</b>
4.2 全连接网络 VS 卷积网络 . .	48	7.2 递归神经网络的前向计算 . .	<b>118</b>
4.3 卷积神经网络是啥 . . . . .	49	7.3 递归神经网络的训练 . . . . .	<b>120</b>
4.4 卷积神经网络输出值的计算	50	7.4 编程实战：递归神经网络的	
4.5 卷积神经网络的训练 . . . .	56	实现 . . . . .	<b>124</b>
4.6 编程实战：卷积神经网络的		7.5 递归神经网络的应用 . . . . .	<b>128</b>
实现 . . . . .	65	7.6 小结 . . . . .	<b>132</b>

# 引言

---

无论即将到来的是大数据时代还是人工智能时代，亦或是传统行业使用人工智能在云上处理大数据的时代，作为一个有理想有追求的程序员，不懂深度学习(Deep Learning)这个超热的技术，会不会感觉马上就 out 了？现在救命稻草来了，《零基础入门深度学习》系列文章旨在讲帮助爱编程的你从零基础达到入门级水平。零基础意味着你不需要太多的数学知识，只要会写程序就行了，没错，这是专门为程序员写的文章。虽然文中会有很多公式你也许看不懂，但同时也会有更多的代码，程序员的你一定能看懂的（我周围是一群狂热的 Clean Code 程序员，所以我写的代码也不会很差）。

这个笔记主要是 Bingtao Han<sup>1</sup> 《零基础入门深度学习》的 LATEX 版本，文中的所有代码都发布[Github](#)上。

---

<sup>1</sup>版权：本文的版权归 Bingtao Han 所有。

# 第1章 感知器

## 内容提要

- 深度学习是啥 1.1
- 感知器 1.2
- 感知器的定义 1.2.1
- 感知器还能做什么 1.2.3
- 编程实战：实现感知器 1.3

## 1.1 深度学习是啥

在人工智能领域，有一个方法叫机器学习。在机器学习这个方法里，有一类算法叫神经网络。神经网络如下图所示：

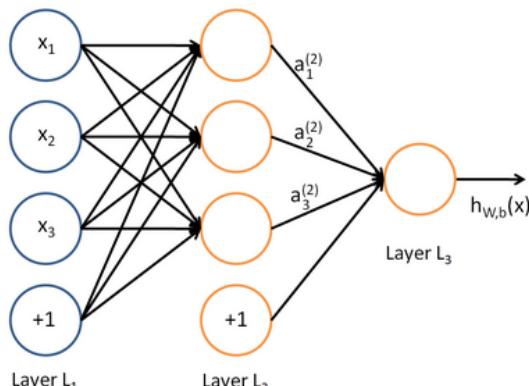


图 1.1：神经网络

图 1.1 中每个圆圈都是一个神经元，每条线表示神经元之间的连接。我们可以看到，上面的神经元被分成了多层，层与层之间的神经元有连接，而层内之间的神经元没有连接。最左边的层叫做输入层，这层负责接收输入数据；最右边的层叫输出层，我们可以从这层获取神经网络输出数据。输入层和输出层之间的层叫做隐藏层。

隐藏层比较多（大于 2）的神经网络叫做深度神经网络。而深度学习，就是使用深层架构（比如，深度神经网络）的机器学习方法。

那么深层网络和浅层网络相比有什么优势呢？简单来说深层网络能够表达力更强。事实上，一个仅有一个隐藏层的神经网络就能拟合任何一个函数，但是它需要很多很多的神经元。而深层网络用少得多的神经元就能拟合同样的函数。也就是为了拟合一个函数，要么使用一个浅而宽的网络，要么使用一个深而窄的网络。而后者往往更节约资源。

深层网络也有劣势，就是它不太容易训练。简单的说，你需要大量的数据，很多的技巧才能训练好一个深层网络。这是个手艺活。

## 1.2 感知器

看到这里，如果你还是一头雾水，那也是很正常的。为了理解神经网络，我们应该先理解神经网络的组成单元——**神经元**。神经元也叫做**感知器**。感知器算法在上个世纪50-70年代很流行，也成功解决了很多问题。并且，感知器算法也是非常简单的。

### 1.2.1 感知器的定义

下图是一个感知器：

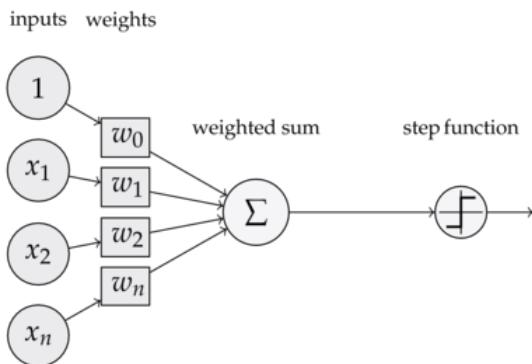


图 1.2: 感知器

可以看到，一个感知器有如下组成部分：

- **输入权值**一个感知器可以接收多个输入  $(x_1, x_2, \dots, x_n \mid x_i \in \mathbb{R})$ ，每个输入上有一个权值  $w_i \in \mathbb{R}$ ，此外还有一个偏置项  $b \in \mathbb{R}$ ，就是上图中  $w_0$ 。
- **激活函数**感知器的激活函数可以有很多选择，比如我们可以选择下面这个阶跃函数  $f$  来作为激活函数：

$$f(x) = \begin{cases} 1 & z > 0 \\ 0 & otherwise \end{cases}$$

- **输出**感知器的输出由下面这个公式来计算

$$y = f(w \bullet x + b) \quad (1.1)$$

如果看完上面的公式一下子就晕了，不要紧，我们用一个简单的例子来帮助理解。

#### 例 1.1 用感知器实现 **and** 函数

我们设计一个感知器，让它来实现 **and** 运算。程序员都知道，**and** 是一个二元函数（带有两个参数  $x_1$  和  $x_2$ ），下面是它的真值表：

表 1.1: **and** 真值表

$x_1$	$x_2$	$y$
0	0	0
0	1	0
1	0	0
1	1	1

为了计算方便，我们用 0 表示 **false**，用 1 表示 **true**。这没什么难理解的，对于 C 语言程序员来说，这是天经地义的。

我们令  $w_1 = 0.5; w_2 = 0.5; b = -0.8$ ，而激活函数  $f$  就是前面写出来的阶跃函数，这时，感知器就相当于**and** 函数。不明白？我们验算一下：

输入上面真值表的第一行，即  $x_1 = 0; x_2 = 0$ ，那么根据公式1.1，计算输出：

$$\begin{aligned} y &= f(w \bullet x + b) = f(w_1 x_1 + w_2 x_2 + b) \\ &= f(0.5 \times 0 + 0.5 \times 0 - 0.8) = f(-0.8) = 0 \end{aligned}$$

也就是当  $x_1, x_2$  都为 0 的时候， $y$  为 0，这就是表1.1的第一行。读者可以自行验证上述真值表的第二、三、四行。

### 例 1.2 用感知器实现 **or** 函数

同样，我们也可以用感知器来实现**or**运算。仅仅需要把偏置项  $b$  的值设置为  $-0.3$  就可以了。我们验算一下，下面是**or**运算的真值表：

表 1.2: **or** 真值表

$x_1$	$x_2$	$y$
0	0	0
0	1	1
1	0	1
1	1	1

我们来验算第二行，这时的输入是  $x_1 = 0; x_2 = 1$ ，带入公式1.1：

$$\begin{aligned} y &= f(w \bullet x + b) = f(w_1 x_1 + w_2 x_2 + b) \\ &= f(0.5 \times 1 + 0.5 \times 0 - 0.3) = f(0.2) = 1 \end{aligned}$$

也就是当  $x_1 = 0; x_2 = 1$  时， $y$  为 1，即表1.2第二行。读者可以自行验证其它行。

## 1.2.2 感知器还能做什么

事实上，感知器不仅仅能实现简单的布尔运算。它可以拟合任何的线性函数，任何线性分类或线性回归问题都可以用感知器来解决。前面的布尔运算可以看作是二分类问题，即给定一个输入，输出 0（属于分类 0）或 1（属于分类 1）。如图1.3(a)所示，**and** 运算是一个线性分类问题，即可以用一条直线把分类 0 (**false**, 红叉表示) 和分类 1 (**true**, 绿点表示) 分开。然而，感知器却不能实现异或运算，如图1.3(b)所示，异或运算不是线性的，你无法用一条直线把分类 0 和分类 1 分开。

## 1.2.3 感知器的训练

现在，你可能困惑前面的权重项和偏置项的值是如何获得的呢？这就要用到感知器训练算法：将权重项和偏置项初始化为 0，然后，利用下面的感知器规则迭代的修改  $w_i$

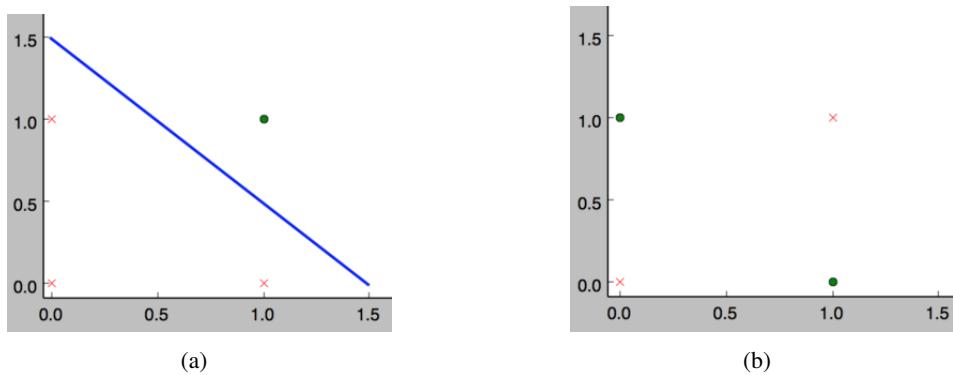


图 1.3: 真值图

和  $b$ , 直到训练完成。

$$w_i \leftarrow w_i + \Delta w_i$$

$$b \leftarrow b + \Delta b$$

其中:

$$\Delta w_i = \eta(t - y)x_i$$

$$\Delta b = \eta(t - y)$$

$w_i$  是与输入  $x_i$  对应的权重项,  $b$  是偏置项。事实上, 可以把  $b$  看作是值永远为 1 的输入  $x_b$  所对应的权重。 $t$  是训练样本的实际值, 一般称之为 **label**。而  $y$  是感知器的输出值, 它是根据公式1.1计算得出。 $\eta$  是一个称为 **学习速率**的常数, 其作用是控制每一步调整权的幅度。

每次从训练数据中取出一个样本的输入向量  $x$ , 使用感知器计算其输出  $y$ , 再根据上面的规则来调整权重。每处理一个样本就调整一次权重。经过多轮迭代后(即全部的训练数据被反复处理多轮), 就可以训练出感知器的权重, 使之实现目标函数。

### 1.3 编程实战：实现感知器



**注意** 完整代码请参考 GitHub: [https://github.com/hanbt/learn\\_dl/blob/master/perceptron.py](https://github.com/hanbt/learn_dl/blob/master/perceptron.py) (python2.7)

对于程序员来说, 没有什么比亲自动手实现学得更快了, 而且, 很多时候一行代码抵得上千言万语。接下来我们就将实现一个感知器。

下面是一些说明:

- 使用 python 语言。python 在机器学习领域用的很广泛, 而且, 写 python 程序真的很轻松。
- 面向对象编程。面向对象是特别好的管理复杂度的工具, 应对复杂问题时, 用面向对象设计方法很容易将复杂问题拆解为多个简单问题, 从而解救我们的大脑。
- 没有使用 numpy。numpy 实现了很多基础算法, 对于实现机器学习算法来说是个必备的工具。但为了降低读者理解的难度, 下面的代码只用到了基本的 python (省去

您去学习 numpy 的时间)。

下面是感知器类的实现，非常简单。去掉注释只有 27 行，而且还包括为了美观（每行不超过 60 个字符）而增加的很多换行。

```
1 class Perceptron(object):
2     def __init__(self, input_num, activator):
3         ...
4         初始化感知器，设置输入参数的个数，以及激活函数。
5         激活函数的类型为 double -> double
6         ...
7         self.activator = activator
8         # 权重向量初始化为 0
9         self.weights = [0.0 for _ in range(input_num)]
10        # 偏置项初始化为 0
11        self.bias = 0.0
12    def __str__(self):
13        ...
14        打印学习到的权重、偏置项
15        ...
16        return 'weights\t:%s\nbias\t:%f\n' % (self.weights, self.
17            bias)
17    def predict(self, input_vec):
18        ...
19        输入向量，输出感知器的计算结果
20        ...
21        # 把 input_vec [x1, x2, x3 ...] 和 weights [w1, w2, w3, ...] 打包在一
22        # 起
23        # 变成 [(x1, w1), (x2, w2), (x3, w3), ...]
24        # 然后利用 map 函数计算 [x1*w1, x2*w2, x3*w3]
25        # 最后利用 reduce 求和
26        return self.activator(
27            reduce(lambda a, b: a + b,
28                  map(lambda (x, w): x * w,
29                      zip(input_vec, self.weights)))
30            , 0.0) + self.bias)
31    def train(self, input_vecs, labels, iteration, rate):
32        ...
33        输入训练数据：一组向量、与每个向量对应的 label；以及训练轮
34        数、学习率
35        ...
36        for i in range(iteration):
37            self._one_iteration(input_vecs, labels, rate)
38    def _one_iteration(self, input_vecs, labels, rate):
39        ...
```

```
38     一次迭代，把所有的训练数据过一遍
39     ...
40     # 把输入和输出打包在一起，成为样本的列表[(input_vec, label
41     # ), ...]
42     # 而每个训练样本是(input_vec, label)
43     samples = zip(input_vecs, labels)
44     # 对每个样本，按照感知器规则更新权重
45     for (input_vec, label) in samples:
46         # 计算感知器在当前权重下的输出
47         output = self.predict(input_vec)
48         # 更新权重
49         self._update_weights(input_vec, output, label, rate)
50     def _update_weights(self, input_vec, output, label, rate):
51         ...
52         按照感知器规则更新权重
53         ...
54         # 把input_vec[x1, x2, x3, ...] 和weights[w1, w2, w3, ...] 打包在一
55         # 起
56         # 变成[(x1, w1), (x2, w2), (x3, w3), ...]
57         # 然后利用感知器规则更新权重
58         delta = label - output
59         self.weights = map(
60             lambda (x, w): w + rate * delta * x,
61             zip(input_vec, self.weights))
62         # 更新bias
63         self.bias += rate * delta
```

接下来，我们利用这个感知器类去实现**and**函数。

```
1 def f(x):
2     ...
3     定义激活函数f
4     ...
5     return 1 if x > 0 else 0
6 def get_training_dataset():
7     ...
8     基于and真值表构建训练数据
9     ...
10    # 构建训练数据
11    # 输入向量列表
12    input_vecs = [[1,1], [0,0], [1,0], [0,1]]
13    # 期望的输出列表，注意要与输入一一对应
14    # [1,1] -> 1, [0,0] -> 0, [1,0] -> 0, [0,1] -> 0
15    labels = [1, 0, 0, 0]
```

```
16     return input_vecs, labels
17 def train_and_perceptron():
18     """
19     使用 and 真值表训练感知器
20     """
21     # 创建感知器，输入参数个数为 2（因为 and 是二元函数），激活函数为
22     # f
23     p = Perceptron(2, f)
24     # 训练，迭代 10 轮，学习速率为 0.1
25     input_vecs, labels = get_training_dataset()
26     p.train(input_vecs, labels, 10, 0.1)
27     # 返回训练好的感知器
28     return p
29 if __name__ == '__main__':
30     # 训练 and 感知器
31     and_perception = train_and_perceptron()
32     # 打印训练获得的权重
33     print and_perception
34     # 测试
35     print '1 and 1 = %d' % and_perception.predict([1, 1])
36     print '0 and 0 = %d' % and_perception.predict([0, 0])
37     print '1 and 0 = %d' % and_perception.predict([1, 0])
38     print '0 and 1 = %d' % and_perception.predict([0, 1])
```

将上述程序保存为 perceptron.py 文件，通过命令行执行这个程序，其运行结果为：

```
hanbingtao-mac:ann hanbingtao$ python perceptron.py
weights :[0.1, 0.2]
bias    :-0.200000

1 and 1 = 1
0 and 0 = 0
1 and 0 = 0
0 and 1 = 0
```

神奇吧！感知器竟然完全实现了 and 函数。读者可以尝试一下利用感知器实现其它函数。

## 1.4 小结

终于看（写）到小结了...，大家都累了。对于零基础的你来说，走到这里应该已经很烧脑了吧。没关系，休息一下。值得高兴的是，你终于已经走出了深度学习入门的第一步，这是巨大的进步；坏消息是，这仅仅是最简单的部分，后面还有无数艰难险阻等着你。不过，你学的困难往往意味着别人学的也困难，掌握一门高门槛的技艺，进可糊口退可装逼，是很值得的。

下一篇文章，我们将讨论另外一种感知器：线性单元，并由此引出一种可能是最重要的优化算法：梯度下降算法。



## 第2章 线性单元和梯度下降

### 内容提要

- 线性单元是啥 2.1
- 线性单元的模型 2.1.1
- 监督学习和无监督学习 2.1.2
- 线性单元的目标函数 2.1.3
- 梯度下降优化算法 2.1.4
- $\nabla E(w)$  的推导 2.1.5
- 随机梯度下降算法 2.1.6
- 编程实战：实现线性单元 2.2

在上一篇文章中，我们已经学会了编写一个简单的感知器，并用它来实现一个线性分类器。你应该还记得用来训练感知器的『感知器规则』。然而，我们并没有关心这个规则是怎么得到的。本文通过介绍另外一种『感知器』，也就是『线性单元』，来说明关于机器学习一些基本的概念，比如模型、目标函数、优化算法等等。这些概念对于所有的机器学习算法来说都是通用的，掌握了这些概念，就掌握了机器学习的基本套路。

### 2.1 线性单元是啥

感知器有一个问题，当面对的数据集不是线性可分的时候，『感知器规则』可能无法收敛，这意味着我们永远也无法完成一个感知器的训练。为了解决这个问题，我们使用一个可导的线性函数来替代感知器的阶跃函数，这种感知器就叫做线性单元。线性单元在面对线性不可分的数据集时，会收敛到一个最佳的近似上。

为了简单起见，我们可以设置线性单元的激活函数  $f$  为  $f(x) = x$ ，这样的线性单元如图2.1所示，对比此前我们讲过的感知器如图2.2所示

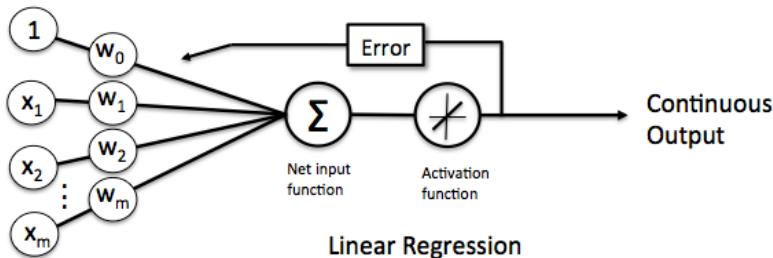


图 2.1：线性单元

这样替换了激活函数  $f$  之后，线性单元将返回一个实数值而不是  $0, 1$  分类。因此线性单元用来解决回归问题而不是分类问题。

#### 2.1.1 线性单元的模型

当我们说模型时，我们实际上在谈论根据输入  $x$  预测输出  $y$  的算法。比如， $x$  可以是一个人的工作年限， $y$  可以是他的月薪，我们可以用某种算法来根据一个人的工作年

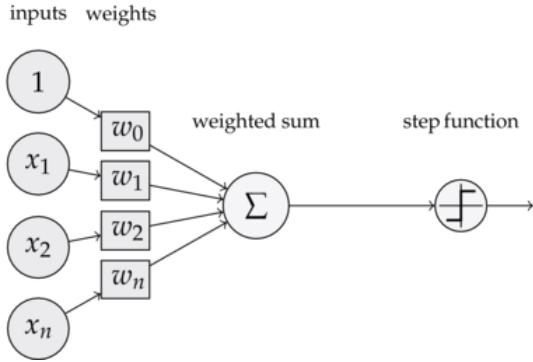


图 2.2: 感知器

限来预测他的收入。比如：

$$y = h(x) = w * x + b$$

函数  $h(x)$  叫做假设，而  $w$ 、 $b$  是它的参数。我们假设参数  $w = 1000$ ，参数  $b = 500$ ，如果一个人的工作年限是 5 年的话，我们的模型会预测他的月薪为

$$y = h(x) = 1000 * 5 + 500 = 5500(\text{元})$$

你也许会说，这个模型太不靠谱了。是这样的，因为我们考虑的因素太少了，仅仅包含了工作年限。如果考虑更多的因素，比如所处的行业、公司、职级等等，可能预测就会靠谱的多。我们把工作年限、行业、公司、职级这些信息，称之为特征。对于一个工作了 5 年，在 IT 行业，百度工作，职级 T6 这样的人，我们可以用这样的一个特征向量来表示他

$$x = (5, \text{IT}, \text{百度}, \text{T6})$$

既然输入  $x$  变成了一个具备四个特征的向量，相对应的，仅仅一个参数  $w$  就不够用了，我们应该使用 4 个参数  $w_1, w_2, w_3, w_4$ ，每个特征对应一个。这样，我们的模型就变成

$$y = h(x) = w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + w_4 * x_4 + b$$

其中， $x_1$  对应工作年限， $x_2$  对应行业， $x_3$  对应公司， $x_4$  对应职级。

为了书写和计算方便，我们可以令  $w_0$  等于  $b$ ，同时令  $w_0$  对应于特征  $x_0$ 。由于  $x_0$  其实并不存在，我们可以令它的值永远为 1。也就是说

$$b = w_0 * x_0 \quad \text{其中 } x_0 = 1$$

这样上面的式子就可以写成

$$\begin{aligned} y = h(x) &= w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + w_4 * x_4 + b \\ &= w_0 * x_0 + w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + w_4 * x_4 \end{aligned}$$

我们还可以把上式写成向量的形式

$$y = h(x) = w^T x \tag{2.1}$$

长成这种样子模型就叫做线性模型，因为输出  $y$  就是输入特征  $x_1, x_2, x_3, \dots$  的线性组合。

## 2.1.2 监督学习和无监督学习

接下来，我们需要关心的是这个模型如何训练，也就是参数  $w$  取什么值最合适。

机器学习有一类学习方法叫做**监督学习**，它是说为了训练一个模型，我们要提供这样一堆训练样本：每个训练样本既包括输入特征  $x$ ，也包括对应的输出  $y$ ( $y$  也叫做**标记**, **label**)。也就是说，我们要找到很多人，我们既知道他们的特征(工作年限，行业...), 也知道他们的收入。我们用这样的样本去训练模型，让模型既看到我们提出的每个问题(输入特征  $x$ )，也看到对应问题的答案(标记  $y$ )。当模型看到足够多的样本之后，它就能总结出其中的一些规律。然后，就可以预测那些它没看过的输入所对应的答案了。

另外一类学习方法叫做**无监督学习**，这种方法的训练样本中只有  $x$  而没有  $y$ 。模型可以总结出特征  $x$  的一些规律，但是无法知道其对应的答案  $y$ 。

很多时候，既有  $x$  又有  $y$  的训练样本是很少的，大部分样本都只有  $x$ 。比如在语音到文本(STT)的识别任务中， $x$  是语音， $y$  是这段语音对应的文本。我们很容易获取大量的语音录音，然而把语音一段一段切分好并标注上对应文字则是非常费力气的事情。这种情况下，为了弥补带标注样本的不足，我们可以用**无监督学习方法**先做一些聚类，让模型总结出哪些音节是相似的，然后再用少量的带标注的训练样本，告诉模型其中一些音节对应的文字。这样模型就可以把相似的音节都对应到相应文字上，完成模型的训练。

## 2.1.3 线性单元的目标函数

现在，让我们只考虑**监督学习**。

在监督学习下，对于一个样本，我们知道它的特征  $x$ ，以及标记  $y$ 。同时，我们还可以根据模型  $h(x)$  计算得到输出  $\bar{y}$ 。注意这里面我们用  $y$  表示训练样本里面的**标记**，也就是**实际值**；用带上划线的  $\bar{y}$  表示模型计算出来的**预测值**。我们当然希望模型计算出来的  $\bar{y}$  和  $y$  越接近越好。

数学上有很多方法来表示的  $\bar{y}$  和  $y$  的接近程度，比如我们可以用  $\bar{y}$  和  $y$  的差的平方的  $\frac{1}{2}$  来表示它们的接近程度

$$e = \frac{1}{2}(y - \bar{y})^2$$

我们把  $e$  叫做**单个样本的误差**。至于为什么前面要乘  $\frac{1}{2}$ ，是为了后面计算方便。

训练数据中会有很多样本，比如  $N$  个，我们可以用训练数据中所有样本的误差的和，来表示模型的误差  $E$ ，也就是

$$E = e^{(1)} + e^{(2)} + e^{(3)} + \dots + e^{(n)}$$

上式的  $e^{(1)}$  表示第一个样本的误差， $e^{(2)}$  表示第二个样本的误差.....。

我们还可以把上面的式子写成和式的形式。使用和式，不光书写起来简单，逼格也跟着暴涨，一举两得。所以一定要写成下面这样

$$E = e^{(1)} + e^{(2)} + e^{(3)} + \dots + e^{(n)} = \sum_{i=1}^n e^{(i)} = \frac{1}{2} \sum_{i=1}^n (y^{(i)} - \bar{y}^{(i)})^2 \quad (2.2)$$

其中

$$\bar{y}^{(i)} = h(x^{(i)}) = w^T x^{(i)}$$

公式2.2中， $x^{(i)}$  表示第  $i$  个训练样本的特征， $y^{(i)}$  表示第  $i$  个样本的标记，我们也可以用元组 $(x^{(i)}, y^{(i)})$  表示第  $i$  训练样本。 $\bar{y}^{(i)}$  则是模型对第  $i$  个样本的预测值。

我们当然希望对于一个训练数据集来说，误差最小越好，也就是公式2.2的值越小越好。对于特定的训练数据集来说， $(x^{(i)}, y^{(i)})$  的值都是已知的，所以公式2.2其实是参数  $w$  的函数。

$$E(w) = \frac{1}{2} \sum_{i=1}^n (y^{(i)} - \bar{y}^{(i)})^2 = \frac{1}{2} \sum_{i=1}^n (y^{(i)} - w^T x^{(i)})^2$$

由此可见，模型的训练，实际上就是求取到合适的  $w$ ，使 (式 2) 取得最小值。这在数学上称作优化问题，而  $E(w)$  就是我们优化的目标，称之为目标函数。

#### 2.1.4 梯度下降优化算法

大学时我们学过怎样求函数的极值。函数  $y = f(x)$  的极值点，就是它的导数  $f'(x) = 0$  的那个点。因此我们可以通过解方程  $f'(x) = 0$ ，求得函数的极值点  $(x_0, y_0)$ 。

不过对于计算机来说，它可不会解方程。但是它可以凭借强大的计算能力，一步一步的去把函数的极值点『试』出来。如图2.3所示：

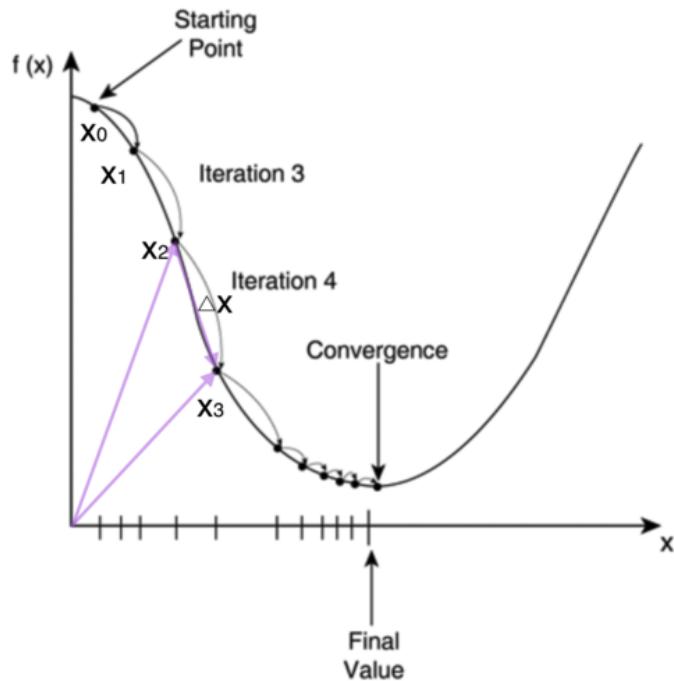


图 2.3: 极值点搜索图

首先，我们随便选择一个点开始，比如上图的  $x_0$  点。接下来，每次迭代修改  $x$  的为  $x_1, x_2, x_3, \dots$ ，经过数次迭代后最终达到函数最小值点。

你可能要问了，为啥每次修改  $x$  的值，都能往函数最小值那个方向前进呢？这里的奥秘在于，我们每次都是向函数  $y = f(x)$  的梯度的相反方向来修改  $x$ 。什么是梯度呢？翻开大学高数课的课本，我们会发现梯度是一个向量，它指向函数值上升最快的方向。显

然，梯度的反方向当然就是函数值下降最快的方向了。我们每次沿着梯度相反方向去修改  $x$  的值，当然就能走到函数的最小值附近。之所以是最小值附近而不是最小值那个点，是因为我们每次移动的步长不会那么恰到好处，有可能最后一次迭代走远了越过了最小值那个点。步长的选择是门手艺，如果选择小了，那么就会迭代很多轮才能走到最小值附近；如果选择大了，那可能就会越过最小值很远，收敛不到一个好的点上。

按照上面的讨论，我们就可以写出梯度下降算法的公式

$$x_{new} = x_{old} - \eta \nabla f(x)$$

其中， $\nabla$  是梯度算子， $\nabla f(x)$  就是指  $f(x)$  的梯度。 $\eta$  是步长，也称作学习速率。

对于上一节列出的目标函数(公式2.2)

$$E(w) = \frac{1}{2} \sum_{i=1}^n (y^{(i)} - \bar{y}^{(i)})^2$$

梯度下降算法可以写成

$$w_{new} = w_{old} - \eta \nabla E(w)$$

聪明的你应该能想到，如果要求目标函数的最大值，那么我们就应该用梯度上升算法，它的参数修改规则是

$$w_{new} = w_{old} + \eta \nabla E(w)$$

下面，请先做几次深呼吸，让你的大脑补充足够的新鲜的氧气，我们要来求取  $\nabla E(w)$ ，然后带入上式，就能得到线性单元的参数修改规则。

关于  $\nabla E(w)$  的推导过程，我单独把它们放到一节中。您既可以选择慢慢看，也可以选择无视。在这里，您只需要知道，经过一大串推导，目标函数  $E(w)$  的梯度是

$$\nabla E(w) = - \sum_{i=1}^n (y^{(i)} - \bar{y}^{(i)}) x^{(i)}$$

因此，线性单元的参数修改规则最后是这个样子

$$w_{new} = w_{old} + \eta \sum_{i=1}^n (y^{(i)} - \bar{y}^{(i)}) x^{(i)} \quad (2.3)$$

有了上面这个式子，我们就可以根据它来写出训练线性单元的代码了。

需要说明的是，如果每个样本有  $M$  个特征，则上式中的  $x, w$  都是  $M+1$  维向量(因为我们加上了一个恒为 1 的虚拟特征  $x_0$ ，参考前面的内容)，而  $y$  是标量。用高逼格的数学符号表示，就是

$$x, w \in \mathbb{R}^{(M+1)}, y \in \mathbb{R}^1$$

为了让您看明白说的是啥，我吐血写下下面这个解释(写这种公式可累可累了)。因

为  $w, x$  是  $M+1$  维列向量，所以 (公式2.3) 可以写成

$$\begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \dots \\ w_m \end{bmatrix}_{new} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \dots \\ w_m \end{bmatrix}_{old} + \eta \sum_{i=1}^n (y^{(i)} - \bar{y}^{(i)}) \begin{bmatrix} 1 \\ x_1^{(i)} \\ x_2^{(i)} \\ \dots \\ x_m^{(i)} \end{bmatrix}$$

如果您还是没看明白，建议您也吐血再看一下大学时学过的《线性代数》吧。

### 2.1.5 $\nabla E(w)$ 的推导

这一节你尽可以跳过它，并不太会影响到全文的理解。当然如果你非要弄明白每个细节，那恭喜你骚年，机器学习的未来一定是属于你的。

首先，我们先做一个简单的前戏。我们知道函数的梯度的定义就是它相对于各个变量的偏导数，所以我们写下下面的式子

$$\nabla E(w) = \frac{\partial}{\partial w} E(w) = \frac{\partial}{\partial w} \frac{1}{2} \sum_{i=1}^n (y^{(i)} - \bar{y}^{(i)})^2$$

可接下来怎么办呢？我们知道和的导数等于导数的和，所以我们可以先把求和符号  $\sum$  里面的导数求出来，然后再把它们加在一起就行了，也就是

$$\frac{\partial}{\partial w} \frac{1}{2} \sum_{i=1}^n (y^{(i)} - \bar{y}^{(i)})^2 = \frac{1}{2} \sum_{i=1}^n \frac{\partial}{\partial w} (y^{(i)} - \bar{y}^{(i)})^2$$

现在我们可以不管高大上的  $\sum$  了，先专心把里面的导数求出来。

$$\frac{\partial}{\partial w} (y^{(i)} - \bar{y}^{(i)})^2 = \frac{\partial}{\partial w} (y^{(i)2} - 2\bar{y}^{(i)}y^{(i)} + \bar{y}^{(i)2})$$

我们知道， $y$  是与  $w$  无关的常数，而  $\bar{y} = w^T x$ ，下面我们根据链式求导法则来求导（上大学时好像叫复合函数求导法则）

$$\frac{\partial E(w)}{\partial w} = \frac{\partial E(\bar{y})}{\partial \bar{y}} \frac{\partial \bar{y}}{\partial w}$$

我们分别计算上式等号右边的两个偏导数

$$\begin{aligned} \frac{\partial E(w)}{\partial \bar{y}} &= \frac{\partial}{\partial \bar{y}} (y^{(i)2} - 2\bar{y}^{(i)}y^{(i)} + \bar{y}^{(i)2}) = -2y^{(i)} + 2\bar{y}^{(i)} \\ \frac{\partial \bar{y}}{\partial w} &= \frac{\partial}{\partial w} w^T x = x \end{aligned}$$

代入，我们求得  $\sum$  里面的偏导数是

$$\frac{\partial}{\partial w} (y^{(i)} - \bar{y}^{(i)})^2 = 2(-y^{(i)} + \bar{y}^{(i)})x$$

最后代入  $\nabla E(w)$ ，求得

$$\nabla E(w) = \frac{1}{2} \sum_{i=1}^n \frac{\partial}{\partial w} (y^{(i)} - \bar{y}^{(i)})^2 = \frac{1}{2} \sum_{i=1}^n 2(-y^{(i)} + \bar{y}^{(i)})x = -\sum_{i=1}^n (y^{(i)} - \bar{y}^{(i)})x$$

至此，大功告成。

### 2.1.6 随机梯度下降算法

如果我们根据(公式2.3)来训练模型，那么我们每次更新 $w$ 的迭代，要遍历训练数据中所有的样本进行计算，我们称这种算法叫做**批梯度下降(Batch Gradient Descent)**。如果我们的样本非常大，比如数百万到数亿，那么计算量异常巨大。因此，实用的算法是SGD算法。在SGD算法中，每次更新 $w$ 的迭代，只计算一个样本。这样对于一个具有数百万样本的训练数据，完成一次遍历就会对 $w$ 更新数百万次，效率大大提升。由于样本的噪音和随机性，每次更新 $w$ 并不一定按照减少 $E$ 的方向。然而，虽然存在一定随机性，大量的更新总体上沿着减少 $E$ 的方向前进的，因此最后也能收敛到最小值附近。下图展示了SGD和BGD的区别

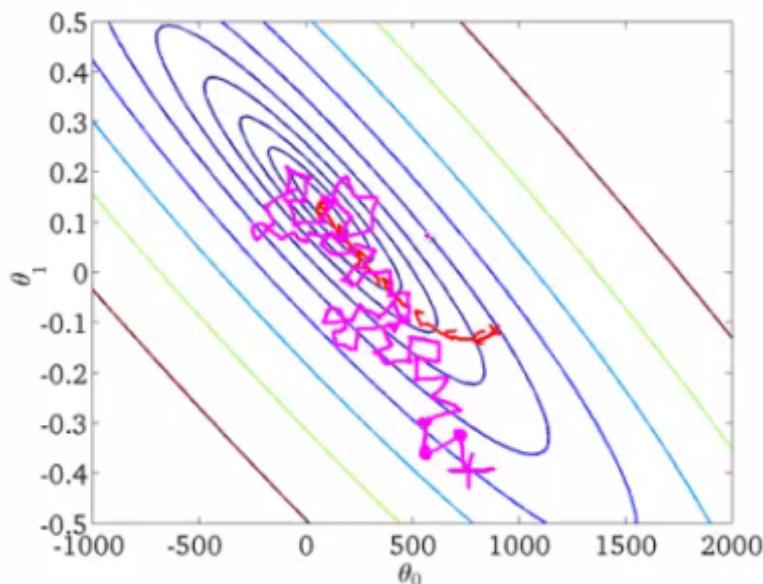


图 2.4: 极值点搜索图

如图2.4，椭圆表示的是函数值的等高线，椭圆中心是函数的最小值点。红色是BGD的逼近曲线，而紫色是SGD的逼近曲线。我们可以看到BGD是一直向着最低点前进的，而SGD明显躁动了许多，但总体上仍然是向最低点逼近的。

最后需要说明的是，SGD不仅效率高，而且随机性有时候反而是好事。今天的目标函数是一个『凸函数』，沿着梯度反方向就能找到全局唯一的最小值。然而对于非凸函数来说，存在许多局部最小值。随机性有助于我们逃离某些很糟糕的局部最小值，从而获得一个更好的模型。

## 2.2 编程实战：实现线性单元



注意 完整代码请参考 GitHub: [https://github.com/hanbt/learn\\_dl/blob/master/unit/unit.py](https://github.com/hanbt/learn_dl/blob/master/unit/unit.py) (python2.7)

接下来，让我们撸一把代码。

因为我们已经写了感知器的代码，因此我们先比较一下感知器模型和线性单元模型，看看哪些代码能够复用。

表 2.1: 模型函数对比

算法	感知器	线性单元
模型 $h(x)$	$y = f(w^T x)$ $f(x) = 1*(z>0)$	$y = f(w^T x)$ $f(z) = z$
训练规则	$w \leftarrow w + \eta(y - \bar{y})x$	$w \leftarrow w + \eta(y - \bar{y})x$

比较的结果令人震惊，原来除了激活函数  $f$  不同之外，两者的模型和训练规则是一样的(在上表中，线性单元的优化算法是 SGD 算法)。那么，我们只需要把感知器的激活函数进行替换即可。感知器的代码请参考第1章，这里就不再重复了。对于一个养成良好习惯的程序员来说，重复代码是不可忍受的。大家应该把代码保存在一个代码库中(比如 git)。

```

1 from perceptron import Perceptron
2 # 定义激活函数f
3 f = lambda x: x
4 class LinearUnit(Perceptron):
5     def __init__(self, input_num):
6         ''' 初始化线性单元，设置输入参数的个数 '''
7         Perceptron.__init__(self, input_num, f)

```

通过继承 Perceptron，我们仅用几行代码就实现了线性单元。这再次证明了面向对象编程范式的强大。

接下来，我们用简单的数据进行一下测试。

```

1 def get_training_dataset():
2     """
3     捏造5个人的收入数据
4     """
5     # 构建训练数据
6     # 输入向量列表，每一项是工作年限
7     input_vecs = [[5], [3], [8], [1.4], [10.1]]
8     # 期望的输出列表，月薪，注意要与输入一一对应
9     labels = [5500, 2300, 7600, 1800, 11400]
10    return input_vecs, labels
11
12 def train_linear_unit():
13     """
14     使用数据训练线性单元
15     """
16     # 创建感知器，输入参数的特征数为1（工作年限）
17     lu = LinearUnit(1)
18     # 训练，迭代10轮，学习速率为0.01
19     input_vecs, labels = get_training_dataset()

```

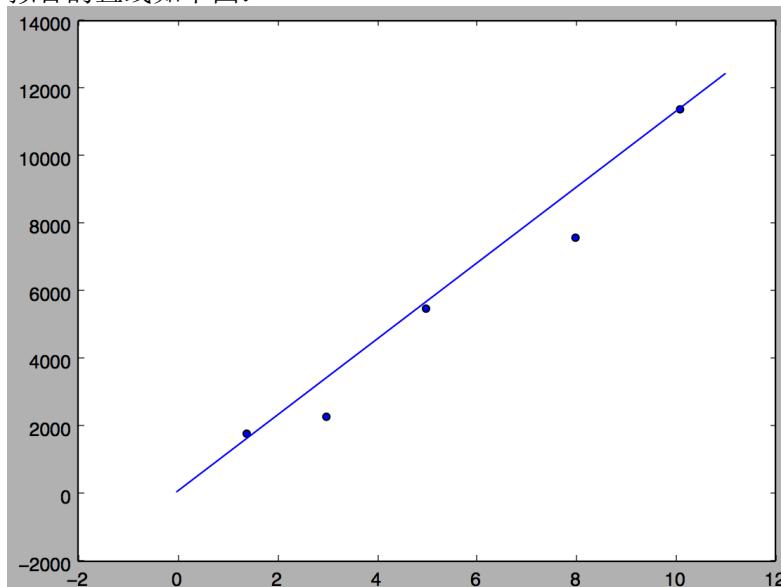
```
19     lu.train(input_vecs, labels, 10, 0.01)
20     # 返回训练好的线性单元
21     return lu
22 if __name__ == '__main__':
23     '''训练线性单元'''
24     linear_unit = train_linear_unit()
25     # 打印训练获得的权重
26     print linear_unit
27     # 测试
28     print 'Work 3.4 years, monthly salary = %.2f' % linear_unit.
29         predict([3.4])
30     print 'Work 15 years, monthly salary = %.2f' % linear_unit.
31         predict([15])
32     print 'Work 1.5 years, monthly salary = %.2f' % linear_unit.
33         predict([1.5])
34     print 'Work 6.3 years, monthly salary = %.2f' % linear_unit.
35         predict([6.3])
```

程序运行结果如下图:

```
hanbingtao-mac:ann hanbingtao$ python linear_unit.py
weights :[1124.0634970262222]
bias      :85.485289

Work 3.4 years, monthly salary = 3907.30
Work 15 years, monthly salary = 16946.44
Work 1.5 years, monthly salary = 1771.58
Work 6.3 years, monthly salary = 7167.09
```

拟合的直线如下图:



## 2.3 小结

事实上，一个机器学习算法其实只有两部分

- 模型从输入特征  $x$  预测输入  $y$  的那个函数  $h(x)$
- 目标函数目标函数取最小(最大)值时所对应的参数值，就是模型的参数的最优值。很多时候我们只能获得目标函数的局部最小(最大)值，因此也只能得到模型参数的局部最优值。

因此，如果你想最简洁的介绍一个算法，列出这两个函数就行了。

接下来，你会用优化算法去求取目标函数的最小(最大)值。**[随机]梯度{下降|上升}**算法就是一个优化算法。针对同一个目标函数，不同的优化算法会推导出不同的训练规则。我们后面还会讲其它的优化算法。

其实在机器学习中，算法往往并不是关键，真正关键之处在于选取特征。选取特征需要我们人类对问题的深刻理解，经验、以及思考。而**神经网络**算法的一个优势，就在于它能够自动学习到应该提取什么特征，从而使算法不再那么依赖人类，而这也是**神经网络**之所以吸引人的一个方面。

现在，经过漫长的烧脑，你已经具备了学习**神经网络**的必备知识。下一篇文文章，我们将介绍本系列文章的主角：**神经网络**，以及用来训练**神经网络**的大名鼎鼎的算法：**反向传播算法**。至于现在，我们应该暂时忘记一切，尽情奖励自己一下吧。



# 第3章 神经网络和反向传播算法

## 内容提要

- 神经元 3.1
- 神经网络是啥 3.2
- 计算神经网络的输出 3.3
- 神经网络的矩阵表示 3.4
- 神经网络的训练 3.5
  - 反向传播算法 3.5.1
  - 反向传播算法的推导 3.5.2
- 编程实战：神经网络的实现 3.6, 3.7.3
- 神经网络实战：手写数字识别 3.7
- 超参数的确定 3.7.1
- 模型的训练和评估 3.7.2
- 向量化编程 3.7.4

在上一篇文章中，我们已经掌握了机器学习的基本套路，对模型、目标函数、优化算法这些概念有了一定程度的理解，而且已经会训练单个的感知器或者线性单元了。在这篇文章中，我们将把这些单独的单元按照一定的规则相互连接在一起形成神经网络，从而奇迹般的获得了强大的学习能力。我们还将介绍这种网络的训练算法：**反向传播算法**。最后，我们依然用代码实现一个神经网络。如果您能坚持到本文的结尾，将会看到我们用自己实现的神经网络去识别手写数字。现在请做好准备，您即将双手触及到深度学习的大门。

## 3.1 神经元

神经元和感知器本质上是一样的，只不过我们说感知器的时候，它的激活函数是阶跃函数；而当我们说神经元时，激活函数往往选择为 sigmoid 函数或 tanh 函数。如图3.1所示：

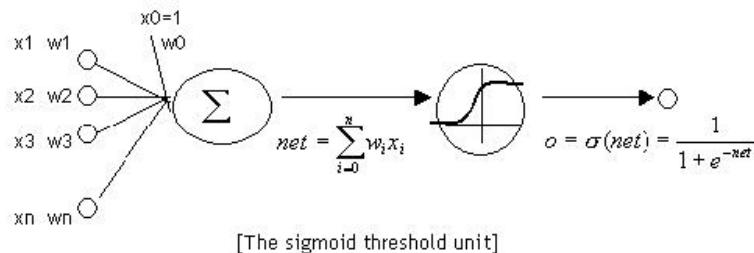


图 3.1: 神经元

计算一个神经元的输出的方法和计算一个感知器的输出是一样的。假设神经元的输入是向量  $\vec{x}$ ，权重向量是  $\vec{w}$ (偏置项是  $w_0$ )，激活函数是 sigmoid 函数，则其输出  $y$ :

$$y = \text{sigmoid}(\vec{w}^T \cdot \vec{x}) \quad (3.1)$$

sigmoid 函数的定义如下：

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

将其带入前面的式子，得到

$$y = \frac{1}{1 + e^{-\vec{w}^T \cdot \vec{x}}}$$

sigmoid 函数是一个非线性函数，值域是 (0,1)。函数图像如图3.2所示

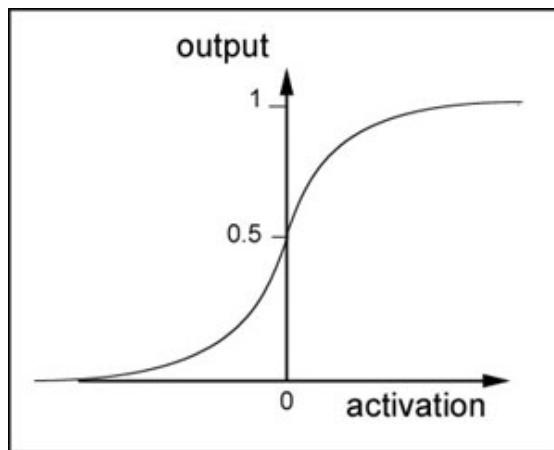


图 3.2: sigmoid 函数

sigmoid 函数的导数是：

$$\text{令 } y = \text{sigmoid}(x)$$

$$\text{则 } y' = y(1 - y)$$

可以看到，sigmoid 函数的导数非常有趣，它可以用 sigmoid 函数自身来表示。这样，一旦计算出 sigmoid 函数的值，计算它的导数的值就非常方便。

## 3.2 神经网络是啥

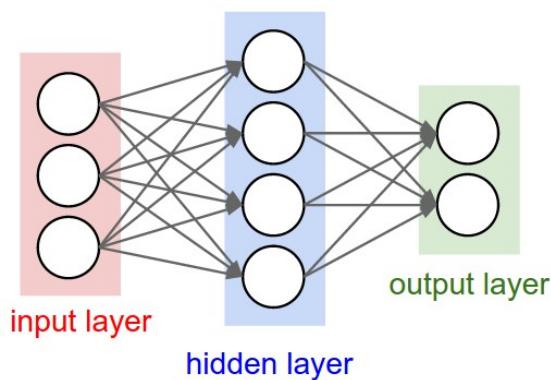


图 3.3: Bp 神经网络

神经网络其实就是按照一定规则连接起来的多个神经元。图3.3展示了一个全连接 (full connected, FC) 神经网络，通过观察可以发现它的规则包括：

- 神经元按照层来布局。最左边的层叫做输入层，负责接收输入数据；最右边的层叫输出层，我们可以从这层获取神经网络输出数据。输入层和输出层之间的层叫做隐藏层，因为它们对于外部来说是不可见的。
- 同一层的神经元之间没有连接。
- 第 N 层的每个神经元和第 N-1 层的所有神经元相连（这就是 full connected 的含义），第 N-1 层神经元的输出就是第 N 层神经元的输入。
- 每个连接都有一个权值。

上面这些规则定义了全连接神经网络的结构。事实上还存在很多其它结构的神经网络，比如卷积神经网络 (CNN)、循环神经网络 (RNN)，他们都具有不同的连接规则。

### 3.3 计算神经网络的输出

神经网络实际上就是一个输入向量  $\vec{x}$  到输出向量  $\vec{y}$  的函数，即：

$$\vec{y} = f_{\text{network}}(\vec{x})$$

根据输入计算神经网络的输出，需要首先将输入向量  $\vec{x}$  的每个元素  $x_i$  的值赋给神经网络的输入层的对应神经元，然后根据公式3.1依次向前计算每一层的每个神经元的值，直到最后一层输出层的所有神经元的值计算完毕。最后，将输出层每个神经元的值串在一起就得到了输出向量  $\vec{y}$ 。

接下来举一个例子来说明这个过程，我们先给神经网络的每个单元写上编号。

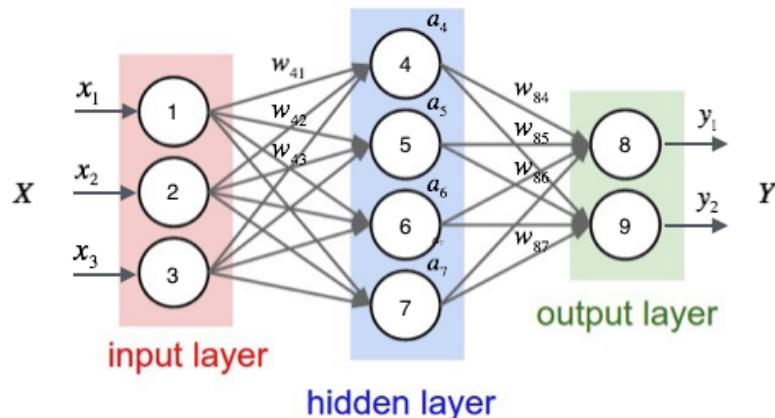


图 3.4: 神经网络

如上图，输入层有三个节点，我们将其依次编号为 1、2、3；隐藏层的 4 个节点，编号依次为 4、5、6、7；最后输出层的两个节点编号为 8、9。因为我们这个神经网络是全连接网络，所以可以看到每个节点都和上一层的所有节点有连接。比如，我们可以看到隐藏层的节点 4，它和输入层的三个节点 1、2、3 之间都有连接，其连接上的权重分别为  $w_{41}, w_{42}, w_{43}$ 。那么，我们怎样计算节点 4 的输出值  $a_4$  呢？

为了计算节点 4 的输出值，我们必须先得到其所有上游节点（也就是节点 1、2、3）的输出值。节点 1、2、3 是输入层的节点，所以，他们的输出值就是输入向量  $\vec{x}$  本身。按照上图画出的对应关系，可以看到节点 1、2、3 的输出值分别是  $x_1, x_2, x_3$ 。我们要求输入

向量的维度和输入层神经元个数相同，而输入向量的某个元素对应到哪个输入节点是可以自由决定的，你偏非要把  $x_1$  赋值给节点 2 也是完全没有问题的，但这样除了把自己弄晕之外，并没有什么价值。

一旦我们有了节点 1、2、3 的输出值，我们就可以根据公式3.1计算节点 4 的输出值  $a_4$ :

$$\begin{aligned} a_4 &= \text{sigmoid}(\vec{w}^T \cdot \vec{x}) \\ &= \text{sigmoid}(w_{41}x_1 + w_{42}x_2 + w_{43}x_3 + w_{4b}) \end{aligned}$$

上式的  $w_{4b}$  是节点 4 的偏置项，图中没有画出来。而  $w_{41}, w_{42}, w_{43}$  分别为节点 1、2、3 到节点 4 连接的权重，在给权重  $w_{ji}$  编号时，我们把目标节点的编号  $j$  放在前面，把源节点的编号  $i$  放在后面。

同样，我们可以继续计算出节点 5、6、7 的输出值  $a_5, a_6, a_7$ 。这样，隐藏层的 4 个节点的输出值就计算完成了，我们就可以接着计算输出层的节点 8 的输出值  $y_1$ :

$$\begin{aligned} y_1 &= \text{sigmoid}(\vec{w}^T \cdot \vec{a}) \\ &= \text{sigmoid}(w_{84}a_4 + w_{85}a_5 + w_{86}a_6 + w_{87}a_7 + w_{8b}) \end{aligned}$$

同理，我们还可以计算出  $y_2$  的值。这样输出层所有节点的输出值计算完毕，我们就得到了在输入向量  $\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$  时，神经网络的输出向量  $\vec{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$ 。这里我们也看到，输出向量的维度和输出层神经元个数相同。

## 3.4 神经网络的矩阵表示

神经网络的计算如果用矩阵来表示会很方便（当然逼格也更高），我们先来看看隐藏层的矩阵表示。

首先我们把隐藏层 4 个节点的计算依次排列出来:

$$\begin{aligned} a_4 &= \text{sigmoid}(w_{41}x_1 + w_{42}x_2 + w_{43}x_3 + w_{4b}) \\ a_5 &= \text{sigmoid}(w_{51}x_1 + w_{52}x_2 + w_{53}x_3 + w_{5b}) \\ a_6 &= \text{sigmoid}(w_{61}x_1 + w_{62}x_2 + w_{63}x_3 + w_{6b}) \\ a_7 &= \text{sigmoid}(w_{71}x_1 + w_{72}x_2 + w_{73}x_3 + w_{7b}) \end{aligned}$$

接着，定义网络的输入向量  $\vec{x}$  和隐藏层每个节点的权重向量  $\vec{w}_j$ 。令

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix}$$

$$\vec{w}_4 = [w_{41}, w_{42}, w_{43}, w_{4b}]$$

$$\vec{w}_5 = [w_{51}, w_{52}, w_{53}, w_{5b}]$$

$$\vec{w}_6 = [w_{61}, w_{62}, w_{63}, w_{6b}]$$

$$\vec{w}_7 = [w_{71}, w_{72}, w_{73}, w_{7b}]$$

$$f = \text{sigmoid}$$

代入到前面的一组式子，得到：

$$a_4 = f(\vec{w}_4 \cdot \vec{x}), \quad a_5 = f(\vec{w}_5 \cdot \vec{x})$$

$$a_6 = f(\vec{w}_6 \cdot \vec{x}), \quad a_7 = f(\vec{w}_7 \cdot \vec{x})$$

现在，我们把上述计算  $a_4, a_5, a_6, a_7$  的四个式子写到一个矩阵里面，每个式子作为矩阵的一行，就可以利用矩阵来表示它们的计算了。令

$$\vec{a} = \begin{bmatrix} a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix}, \quad W = \begin{bmatrix} \vec{w}_4 \\ \vec{w}_5 \\ \vec{w}_6 \\ \vec{w}_7 \end{bmatrix} = \begin{bmatrix} w_{41}, w_{42}, w_{43}, w_{4b} \\ w_{51}, w_{52}, w_{53}, w_{5b} \\ w_{61}, w_{62}, w_{63}, w_{6b} \\ w_{71}, w_{72}, w_{73}, w_{7b} \end{bmatrix}, \quad f\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \end{bmatrix}\right) = \begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ \vdots \end{bmatrix}$$

带入前面的一组式子，得到

$$\vec{a} = f(W \cdot \vec{x}) \quad (3.2)$$

在公式3.2中， $f$  是激活函数，在本例中是 sigmoid 函数； $W$  是某一层的权重矩阵； $\vec{x}$  是某层的输入向量； $\vec{a}$  是某层的输出向量。公式3.2说明神经网络的每一层的作用实际上就是先将输入向量左乘一个数组进行线性变换，得到一个新的向量，然后再对这个向量逐元素应用一个激活函数。

每一层的算法都是一样的。比如，对于包含一个输入层，一个输出层和三个隐藏层的神经网络，我们假设其权重矩阵分别为  $W_1, W_2, W_3, W_4$ ，每个隐藏层的输出分别是  $\vec{a}_1, \vec{a}_2, \vec{a}_3$ ，神经网络的输入为  $\vec{x}$ ，神经网络的输出为  $\vec{y}$ ，如图3.5所示：

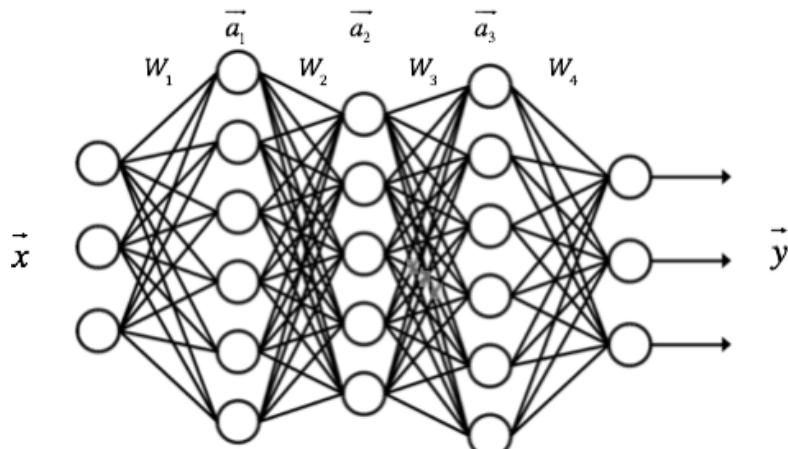


图 3.5：神经网络

则每一层的输出向量的计算可以表示为：

$$\begin{aligned}\vec{a}_1 &= f(W_1 \cdot \vec{x}), & \vec{a}_2 &= f(W_2 \cdot \vec{a}_1) \\ \vec{a}_3 &= f(W_3 \cdot \vec{a}_2), & \vec{y} &= f(W_4 \cdot \vec{a}_3)\end{aligned}$$

这就是神经网络输出值的计算方法。

## 3.5 神经网络的训练

现在，我们需要知道一个神经网络的每个连接上的权值是如何得到的。我们可以说神经网络是一个模型，那么这些权值就是模型的参数，也就是模型要学习的东西。然而，一个神经网络的连接方式、网络的层数、每层的节点数这些参数，则不是学习出来的，而是人为事先设置的。对于这些人为设置的参数，我们称之为超参数 (Hyper-Parameters)。

接下来，我们将要介绍神经网络的训练算法：反向传播算法。

### 3.5.1 反向传播算法 (Back Propagation)

我们首先直观的介绍反向传播算法，最后再来介绍这个算法的推导。当然读者也可以完全跳过推导部分，因为即使不知道如何推导，也不影响你写出来一个神经网络的训练代码。事实上，现在神经网络成熟的开源实现多如牛毛，除了练手之外，你可能都没有机会需要去写一个神经网络。

我们以监督学习为例来解释反向传播算法。在第2章中我们介绍了什么是监督学习，如果忘了可以再看一下。另外，我们设神经元的激活函数  $f$  为 sigmoid 函数 (不同激活函数的计算公式不同，详情见3.5.2反向传播算法的推导一节)。

我们假设每个训练样本为  $(\vec{x}, \vec{t})$ ，其中向量  $\vec{x}$  是训练样本的特征，而  $\vec{t}$  是样本的目标值。

首先，我们根据上一节介绍的算法，用样本的特征  $\vec{x}$ ，计算出神经网络中每个隐藏层节点的输出  $a_i$ ，以及输出层每个节点的输出  $y_i$ 。

然后，我们按照下面的方法计算出每个节点的误差项  $\delta_i$ ：

- 对于输出层节点  $i$ ，

$$\delta_i = y_i(1 - y_i)(t_i - y_i) \quad (3.3)$$

其中， $\delta_i$  是节点  $i$  的误差项， $y_i$  是节点  $i$  的输出值， $t_i$  是样本对应于节点  $i$  的目标值。举个例子，根据上图，对于输出层节点 8 来说，它的输出值是  $y_1$ ，而样本的目标值是  $t_1$ ，带入上面的公式得到节点 8 的误差项  $\delta_8$  应该是：

$$\delta_8 = y_1(1 - y_1)(t_1 - y_1)$$

- 对于隐藏层节点，

$$\delta_i = a_i(1 - a_i) \sum_{k \in outputs} w_{ki} \delta_k \quad (3.4)$$

其中， $a_i$  是节点  $i$  的输出值， $w_{ki}$  是节点  $i$  到它的下一层节点  $k$  的连接的权重， $\delta_k$  是节点

$i$  的下一层节点  $k$  的误差项。例如，对于隐藏层节点 4 来说，计算方法如下：

$$\delta_4 = a_4(1 - a_4)(w_{84}\delta_8 + w_{94}\delta_9)$$

最后，更新每个连接上的权值：

$$w_{ji} \leftarrow w_{ji} + \eta \delta_j x_{ji} \quad (3.5)$$

其中， $w_{ji}$  是节点  $i$  到节点  $j$  的权重， $\eta$  是一个成为学习速率的常数， $\delta_j$  是节点的误差项， $x_{ji}$  是节点  $i$  传递给节点  $j$  的输入。例如，权重  $w_{84}$  的更新方法如下：

$$w_{84} \leftarrow w_{84} + \eta \delta_8 a_4$$

类似的，权重  $w_{41}$  的更新方法如下：

$$w_{41} \leftarrow w_{41} + \eta \delta_4 x_1$$

偏置项的输入值永远为 1。例如，节点 4 的偏置项  $w_{4b}$  应该按照下面的方法计算：

$$w_{4b} \leftarrow w_{4b} + \eta \delta_4$$

我们已经介绍了神经网络每个节点误差项的计算和权重更新方法。显然，计算一个节点的误差项，需要先计算每个与其相连的下一层节点的误差项。这就要求误差项的计算顺序必须是从输出层开始，然后反向依次计算每个隐藏层的误差项，直到与输入层相连的那个隐藏层。这就是反向传播算法的名字的含义。当所有节点的误差项计算完毕后，我们就可以根据公式3.5来更新所有的权重。

以上就是基本的反向传播算法，并不是很复杂，您弄清楚了么？

### 3.5.2 反向传播算法的推导

反向传播算法其实就是链式求导法则的应用。然而，这个如此简单且显而易见的方法，却是在 Roseblatt 提出感知器算法将近 30 年之后才被发明和普及的。对此，Bengio 这样回应道：

 **注意** 很多看似显而易见的想法只有在事后才变得显而易见。

接下来，我们用链式求导法则来推导反向传播算法，也就是上一小节的公式3.3, 3.4, 3.5。

按照机器学习的通用套路，我们先确定神经网络的目标函数，然后用随机梯度下降优化算法去求目标函数最小值时的参数值。

我们取网络所有输出层节点的误差平方和作为目标函数：

$$E_d \equiv \frac{1}{2} \sum_{i \in outputs} (t_i - y_i)^2$$

其中， $E_d$  表示是样本  $d$  的误差。

然后，我们用第2章中介绍的随机梯度下降算法对目标函数进行优化：

$$w_{ji} \leftarrow w_{ji} - \eta \frac{\partial E_d}{\partial w_{ji}}$$

随机梯度下降算法也就是需要求出误差  $E_d$  对于每个权重  $w_{ji}$  的偏导数（也就是梯度），怎么求呢？观察图3.4，我们发现权重  $w_{ji}$  仅能通过影响节点  $j$  的输入值影响网络的

其它部分，设  $net_j$  是节点  $j$  的加权输入，即

$$net_j = \vec{w}_j \cdot \vec{x}_j = \sum_i w_{ji} x_{ji}$$

$E_d$  是  $net_j$  的函数，而  $net_j$  是  $w_{ji}$  的函数。根据链式求导法则，可以得到：

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial \sum_i w_{ji} x_{ji}}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} x_{ji}$$

上式中， $x_{ji}$  是节点  $i$  传递给节点  $j$  的输入值，也就是节点  $i$  的输出值。

对于  $\frac{\partial E_d}{\partial net_j}$  的推导，需要区分输出层和隐藏层两种情况。

### 输出层权值训练

对于输出层来说， $net_j$  仅能通过节点  $j$  的输出值  $y_j$  来影响网络其它部分，也就是说  $E_d$  是  $y_j$  的函数，而  $y_j$  是  $net_j$  的函数，其中  $y_j = sigmoid(net_j)$ 。所以我们可以再次使用链式求导法则：

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial y_j} \frac{\partial y_j}{\partial net_j}$$

考虑上式第一项：

$$\begin{aligned} \frac{\partial E_d}{\partial y_j} &= \frac{\partial}{\partial y_j} \frac{1}{2} \sum_{i \in outputs} (t_i - y_i)^2 \\ &= \frac{\partial}{\partial y_j} \frac{1}{2} (t_j - y_j)^2 \\ &= -(t_j - y_j) \end{aligned}$$

考虑上式第二项：

$$\begin{aligned} \frac{\partial y_j}{\partial net_j} &= \frac{\partial sigmoid(net_j)}{\partial net_j} \\ &= y_j(1 - y_j) \end{aligned}$$

将第一项和第二项带入，得到：

$$\frac{\partial E_d}{\partial net_j} = -(t_j - y_j)y_j(1 - y_j)$$

如果令  $\delta_j = -\frac{\partial E_d}{\partial net_j}$ ，也就是一个节点的误差项  $\delta$  是网络误差对这个节点输入的偏导数的相反数。带入上式，得到：

$$\delta_j = (t_j - y_j)y_j(1 - y_j)$$

上式就是公式3.3。

将上述推导带入随机梯度下降公式，得到：

$$\begin{aligned} w_{ji} &\leftarrow w_{ji} - \eta \frac{\partial E_d}{\partial w_{ji}} \\ &= w_{ji} + \eta(t_j - y_j)y_j(1 - y_j)x_{ji} \\ &= w_{ji} + \eta\delta_j x_{ji} \end{aligned}$$

上式就是公式3.5。

### 隐藏层权值训练

现在我们要推导出隐藏层的  $\frac{\partial E_d}{\partial net_j}$ 。

首先，我们需要定义节点  $j$  的所有直接下游节点的集合  $Downstream(j)$ 。例如，对于节点 4 来说，它的直接下游节点是节点 8、节点 9。可以看到  $net_j$  只能通过影响  $Downstream(j)$  再影响  $E_d$ 。设  $net_k$  是节点  $j$  的下游节点的输入，则  $E_d$  是  $net_k$  的函数，而  $net_k$  是  $net_j$  的函数。因为  $net_k$  有多个，我们应用全导数公式，可以做出如下推导：

$$\begin{aligned}\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial a_j} \frac{\partial a_j}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k w_{kj} \frac{\partial a_j}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k w_{kj} a_j (1 - a_j) \\ &= -a_j (1 - a_j) \sum_{k \in Downstream(j)} \delta_k w_{kj}\end{aligned}$$

因为  $\delta_j = -\frac{\partial E_d}{\partial net_j}$ ，带入上式得到：

$$\delta_j = a_j (1 - a_j) \sum_{k \in Downstream(j)} \delta_k w_{kj}$$

上式就是公式3.4。

#### ——数学公式警报解除——

至此，我们已经推导出了反向传播算法。需要注意的是，我们刚刚推导出的训练规则是根据激活函数是 sigmoid 函数、平方和误差、全连接网络、随机梯度下降优化算法。如果激活函数不同、误差计算方式不同、网络连接结构不同、优化算法不同，则具体的训练规则也会不一样。但是无论怎样，训练规则的推导方式都是一样的，应用链式求导法则进行推导即可。

## 3.6 编程实战：神经网络的实现



**注意** 完整代码请参考 GitHub:[https://github.com/hanbt/learn\\_dl/blob/master/bp.py](https://github.com/hanbt/learn_dl/blob/master/bp.py) (python2.7)

现在，我们要根据前面的算法，实现一个基本的全连接神经网络，这并不需要太多代码。我们在这里依然采用面向对象设计。

首先，我们先做一个基本的模型：

如图3.6，可以分解出 5 个领域对象来实现神经网络：

- *Network* 神经网络对象，提供 API 接口。它由若干层对象组成以及连接对象组成。
- *Layer* 层对象，由多个节点组成。
- *Node* 节点对象计算和记录节点自身的信息（比如输出值  $a$ 、误差项  $\delta$  等），以及与这个节点相关的上下游的连接。

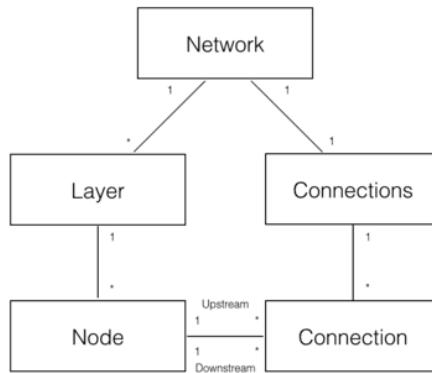


图 3.6: 基础模型

- *Connection* 每个连接对象都要记录该连接的权重。
- *Connections* 仅仅作为 *Connection* 的集合对象，提供一些集合操作。

*Node* 实现如下：

```

1 # 节点类，负责记录和维护节点自身信息以及与这个节点相关的上下游连
2     接，实现输出值和误差项的计算。
3
4 class Node(object):
5     def __init__(self, layer_index, node_index):
6         ...
7         构造节点对象。
8
9         layer_index: 节点所属的层的编号
10        node_index: 节点的编号
11        ...
12
13        self.layer_index = layer_index
14        self.node_index = node_index
15        self.downstream = []
16        self.upstream = []
17        self.output = 0
18        self.delta = 0
19
20    def set_output(self, output):
21        ...
22        设置节点的输出值。如果节点属于输入层会用到这个函数。
23        ...
24
25    self.output = output
26
27    def append_downstream_connection(self, conn):
28        ...
29        添加一个到下游节点的连接
30        ...
31
32        self.downstream.append(conn)
33
34    def append_upstream_connection(self, conn):
35        ...
36        添加一个到上游节点的连接
37        ...
  
```

```

29         self.upstream.append(conn)
30     def calc_output(self):
31         ...
32         根据式1计算节点的输出
33         ...
34         output = reduce(lambda ret, conn: ret + conn.upstream_node
35                         .output * conn.weight, self.upstream, 0)
36         self.output = sigmoid(output)
37     def calc_hidden_layer_delta(self):
38         ...
39         节点属于隐藏层时，根据式4计算delta
40         ...
41         downstream_delta = reduce(
42             lambda ret, conn: ret + conn.downstream_node.delta *
43                 conn.weight,
44                 self.downstream, 0.0)
45         self.delta = self.output * (1 - self.output) *
46             downstream_delta
47     def calc_output_layer_delta(self, label):
48         ...
49         节点属于输出层时，根据式3计算delta
50         ...
51         self.delta = self.output * (1 - self.output) * (label -
52             self.output)
53     def __str__(self):
54         ...
55         打印节点的信息
56         ...
57         node_str = '%u-%u: output: %f delta: %f' % (self.
58             layer_index, self.node_index, self.output, self.delta)
59         downstream_str = reduce(lambda ret, conn: ret + '\n\t' +
60             str(conn), self.downstream, '')
61         upstream_str = reduce(lambda ret, conn: ret + '\n\t' + str
62             (conn), self.upstream, '')
63     return node_str + '\n\tdownstream:' + downstream_str + '\n
64             \tupstream:' + upstream_str

```

ConstNode 对象，为了实现一个输出恒为 1 的节点(计算偏置项  $w_b$  时需要)

```

1 class ConstNode(object):
2     def __init__(self, layer_index, node_index):
3         ...
4         构造节点对象。
5         layer_index: 节点所属的层的编号

```

```
6     node_index: 节点的编号
7     ...
8     self.layer_index = layer_index
9     self.node_index = node_index
10    self.downstream = []
11    self.output = 1
12    def append_downstream_connection(self, conn):
13        ...
14        添加一个到下游节点的连接
15        ...
16        self.downstream.append(conn)
17    def calc_hidden_layer_delta(self):
18        ...
19        节点属于隐藏层时，根据式4计算delta
20        ...
21        downstream_delta = reduce(
22            lambda ret, conn: ret + conn.downstream_node.delta *
23            conn.weight,
24            self.downstream, 0.0)
25        self.delta = self.output * (1 - self.output) *
26            downstream_delta
27    def __str__(self):
28        ...
29        打印节点的信息
30        ...
31        node_str = '%u-%u: output: 1' % (self.layer_index, self.
32            node_index)
33        downstream_str = reduce(lambda ret, conn: ret + '\n\t' +
34            str(conn), self.downstream, '')
35        return node_str + '\n\tdownstream:' + downstream_str
```

Layer 对象，负责初始化一层。此外，作为 Node 的集合对象，提供对 Node 集合的操作。

```
1 class Layer(object):
2     def __init__(self, layer_index, node_count):
3         ...
4         初始化一层
5         layer_index: 层编号
6         node_count: 层所包含的节点个数
7         ...
8         self.layer_index = layer_index
9         self.nodes = []
10        for i in range(node_count):
```

```
11         self.nodes.append(Node(layer_index, i))
12         self.nodes.append(ConstNode(layer_index, node_count))
13     def set_output(self, data):
14         ...
15         设置层的输出。当层是输入层时会用到。
16         ...
17         for i in range(len(data)):
18             self.nodes[i].set_output(data[i])
19     def calc_output(self):
20         ...
21         计算层的输出向量
22         ...
23         for node in self.nodes[:-1]:
24             node.calc_output()
25     def dump(self):
26         ...
27         打印层的信息
28         ...
29         for node in self.nodes:
30             print node
```

Connection 对象，主要职责是记录连接的权重，以及这个连接所关联的上下游节点。

```
1 class Connection(object):
2     def __init__(self, upstream_node, downstream_node):
3         ...
4         初始化连接，权重初始化为是一个很小的随机数
5         upstream_node: 连接的上游节点
6         downstream_node: 连接的下游节点
7         ...
8         self.upstream_node = upstream_node
9         self.downstream_node = downstream_node
10        self.weight = random.uniform(-0.1, 0.1)
11        self.gradient = 0.0
12    def calc_gradient(self):
13        ...
14        计算梯度
15        ...
16        self.gradient = self.downstream_node.delta * self.
17                      upstream_node.output
17    def get_gradient(self):
18        ...
19        获取当前的梯度
20        ...
```

```
21     return self.gradient
22 def update_weight(self, rate):
23     ...
24     根据梯度下降算法更新权重
25     ...
26     self.calc_gradient()
27     self.weight += rate * self.gradient
28 def __str__(self):
29     ...
30     打印连接信息
31     ...
32     return '(%u-%u) -> (%u-%u) = %f' % (
33         self.upstream_node.layer_index,
34         self.upstream_node.node_index,
35         self.downstream_node.layer_index,
36         self.downstream_node.node_index,
37         self.weight)
```

Connections 对象，提供 Connection 集合操作。

```
1 class Connections(object):
2     def __init__(self):
3         self.connections = []
4     def add_connection(self, connection):
5         self.connections.append(connection)
6     def dump(self):
7         for conn in self.connections:
8             print conn
```

Network 对象，提供 API。

```
1 class Network(object):
2     def __init__(self, layers):
3         ...
4         初始化一个全连接神经网络
5         layers: 二维数组，描述神经网络每层节点数
6         ...
7         self.connections = Connections()
8         self.layers = []
9         layer_count = len(layers)
10        node_count = 0;
11        for i in range(layer_count):
12            self.layers.append(Layer(i, layers[i]))
13        for layer in range(layer_count - 1):
14            connections = [Connection(upstream_node,
```



```
15             for upstream_node in self.layers[layer]
16                 .nodes
17
18                 for downstream_node in self.layers[
19                     layer + 1].nodes[:-1]
20
21                     for conn in connections:
22
23                         self.connections.add_connection(conn)
24                         conn.downstream_node.append_upstream_connection(
25                             conn)
26                         conn.upstream_node.append_downstream_connection(
27                             conn)
28
29     def train(self, labels, data_set, rate, iteration):
30         ...
31
32         训练神经网络
33
34         labels: 数组，训练样本标签。每个元素是一个样本的标签。
35         data_set: 二维数组，训练样本特征。每个元素是一个样本的特
36         征。
37
38         ...
39
40         for i in range(iteration):
41             for d in range(len(data_set)):
42                 self.train_one_sample(labels[d], data_set[d], rate)
43
44     def train_one_sample(self, label, sample, rate):
45         ...
46
47         内部函数，用一个样本训练网络
48
49         ...
50
51         self.predict(sample)
52         self.calc_delta(label)
53         self.update_weight(rate)
54
55     def calc_delta(self, label):
56
57         ...
58
59         内部函数，计算每个节点的 delta
60
61         ...
62
63         output_nodes = self.layers[-1].nodes
64         for i in range(len(label)):
65             output_nodes[i].calc_output_layer_delta(label[i])
66
67         for layer in self.layers[-2::-1]:
68             for node in layer.nodes:
69                 node.calc_hidden_layer_delta()
70
71     def update_weight(self, rate):
72
73         ...
74
75         内部函数，更新每个连接权重
76
77         ...
78
79         for layer in self.layers[:-1]:
```

```
52         for node in layer.nodes:
53             for conn in node.downstream:
54                 conn.update_weight(rate)
55     def calc_gradient(self):
56         ...
57         内部函数，计算每个连接的梯度
58         ...
59         for layer in self.layers[:-1]:
60             for node in layer.nodes:
61                 for conn in node.downstream:
62                     conn.calc_gradient()
63     def get_gradient(self, label, sample):
64         ...
65         获得网络在一个样本下，每个连接上的梯度
66         label: 样本标签
67         sample: 样本输入
68         ...
69         self.predict(sample)
70         self.calc_delta(label)
71         self.calc_gradient()
72     def predict(self, sample):
73         ...
74         根据输入的样本预测输出值
75         sample: 数组，样本的特征，也就是网络的输入向量
76         ...
77         self.layers[0].set_output(sample)
78         for i in range(1, len(self.layers)):
79             self.layers[i].calc_output()
80         return map(lambda node: node.output, self.layers[-1].nodes
81                    [-1])
82     def dump(self):
83         ...
84         打印网络信息
85         ...
86         for layer in self.layers:
87             layer.dump()
```

至此，实现了一个基本的全连接神经网络。可以看到，同神经网络的强大学习能力相比，其实现还算是很容易的。

### 梯度检查

怎么保证自己写的神经网络没有 BUG 呢？事实上这是一个非常重要的问题。一方面，千辛万苦想到一个算法，结果效果不理想，那么是算法本身错了还是代码实现错了呢？定位这种问题肯定要花费大量的时间和精力。另一方面，由于神经网络的复杂性，我

们几乎无法事先知道神经网络的输入和输出，因此类似 TDD(测试驱动开发)这样的开发方法似乎也不可行。

办法还是有滴，就是利用梯度检查来确认程序是否正确。梯度检查的思路如下：

对于梯度下降算法：

$$w_{ji} \leftarrow w_{ji} - \eta \frac{\partial E_d}{\partial w_{ji}}$$

来说，这里关键之处在子  $\frac{\partial E_d}{\partial w_{ji}}$  的计算一定要正确，而它是  $E_d$  对  $w_{ji}$  的偏导数。而根据导数的定义：

$$f'(\theta) = \lim_{\epsilon \rightarrow 0} \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon}$$

对于任意  $\theta$  的导数值，我们都可以用等式右边来近似计算。我们把  $E_d$  看做是  $w_{ji}$  的函数，即  $E_d(w_{ji})$ ，那么根据导数定义， $\frac{\partial E_d(w_{ji})}{\partial w_{ji}}$  应该等于：

$$\frac{\partial E_d(w_{ji})}{\partial w_{ji}} = \lim_{\epsilon \rightarrow 0} \frac{f(w_{ji} + \epsilon) - f(w_{ji} - \epsilon)}{2\epsilon}$$

如果把  $\epsilon$  设置为一个很小的数（比如  $10^{-4}$ ），那么上式可以写成：

$$\frac{\partial E_d(w_{ji})}{\partial w_{ji}} \approx \frac{f(w_{ji} + \epsilon) - f(w_{ji} - \epsilon)}{2\epsilon} \quad (3.6)$$

我们就可以利用公式3.6，来计算梯度  $\frac{\partial E_d}{\partial w_{ji}}$  的值，然后同我们神经网络代码中计算出来的梯度值进行比较。如果两者的差别非常的小，那么就说明我们的代码是正确的。

下面是梯度检查的代码。如果我们想检查参数  $w_{ji}$  的梯度是否正确，我们需要以下几个步骤：

1. 首先使用一个样本  $d$  对神经网络进行训练，这样就能获得每个权重的梯度。
2. 将  $w_{ji}$  加上一个很小的值 ( $10^{-4}$ )，重新计算神经网络在这个样本  $d$  下的  $E_{d+}$ 。
3. 将  $w_{ji}$  减上一个很小的值 ( $10^{-4}$ )，重新计算神经网络在这个样本  $d$  下的  $E_{d-}$ 。
4. 根据公式3.6计算出期望的梯度值，和第一步获得的梯度值进行比较，它们应该几乎相等（至少 4 位有效数字相同）。

当然，我们可以重复上面的过程，对每个权重  $w_{ji}$  都进行检查。也可以使用多个样本重复检查。

```

1 def gradient_check(network, sample_feature, sample_label):
2     ...
3     梯度检查
4     network: 神经网络对象
5     sample_feature: 样本的特征
6     sample_label: 样本的标签
7     ...
8     # 计算网络误差
9     network_error = lambda vec1, vec2: \
10         0.5 * reduce(lambda a, b: a + b,
11                     map(lambda v: (v[0] - v[1]) * (v[0] - v[1]),
12                         zip(vec1, vec2)))
13     # 获取网络在当前样本下每个连接的梯度

```

```
14     network.get_gradient(sample_feature, sample_label)
15     # 对每个权重做梯度检查
16     for conn in network.connections.connections:
17         # 获取指定连接的梯度
18         actual_gradient = conn.get_gradient()
19         # 增加一个很小的值，计算网络的误差
20         epsilon = 0.0001
21         conn.weight += epsilon
22         error1 = network_error(network.predict(sample_feature),
23                                sample_label)
23         # 减去一个很小的值，计算网络的误差
24         conn.weight -= 2 * epsilon # 刚才加过了一次，因此这里需要
25             # 减去2倍
25         error2 = network_error(network.predict(sample_feature),
26                                sample_label)
26         # 根据式6计算期望的梯度值
27         expected_gradient = (error2 - error1) / (2 * epsilon)
28         # 打印
29         print 'expected gradient: %f\nactual gradient: %f' %
30             (expected_gradient, actual_gradient)
```

至此，会推导、会实现、会抓 BUG，你已经摸到深度学习的大门了。接下来还需要不断的实践，我们用刚刚写过的神经网络去识别手写数字。

## 3.7 神经网络实战：手写数字识别

针对这个任务，我们采用业界非常流行的 MNIST 数据集。MNIST 大约有 60000 个手写字母的训练样本，我们使用它训练我们的神经网络，然后再用训练好的网络去识别手写数字。

手写数字识别是个比较简单的任务，数字只可能是 0-9 中的一个，这是个 10 分类问题。

### 3.7.1 超参数的确定

我们首先需要确定网络的层数和每层的节点数。关于第一个问题，实际上并没有什么理论化的方法，大家都是根据经验来拍，如果没有经验的话就随便拍一个。然后，你可以多试几个值，训练不同层数的神经网络，看看哪个效果最好就用哪个。嗯，现在你可能明白为什么说深度学习是个手艺活了，有些手艺很让人无语，而有些手艺还是很有技术含量的。

不过，有些基本道理我们还是明白的，我们知道网络层数越多越好，也知道层数越多训练难度越大。对于全连接网络，隐藏层最好不要超过三层。那么，我们可以先试试仅有一个隐藏层的神经网络效果怎么样。毕竟模型小的话，训练起来也快些(刚开始玩模

型的时候，都希望快点看到结果)。

输入层节点数是确定的。因为 MNIST 数据集每个训练数据是 28\*28 的图片，共 784 个像素，因此，输入层节点数应该是 784，每个像素对应一个输入节点。

输出层节点数也是确定的。因为是 10 分类，我们可以用 10 个节点，每个节点对应一个分类。输出层 10 个节点中，输出最大值的那个节点对应的分类，就是模型的预测结果。

隐藏层节点数量是不好确定的，从 1 到 100 万都可以。下面有几个经验公式：

$$m = \sqrt{n + l} + \alpha$$

$$m = \log_2 n$$

$$m = \sqrt{nl}$$

其中， $m$ : 隐藏层节点数； $n$ : 输入层节点数； $l$ : 输出层节点数； $\alpha$ : 1 到 10 之间的常数。因此，我们可以先根据上面的公式设置一个隐藏层节点数。如果有时间，我们可以设置不同的节点数，分别训练，看看哪个效果最好就用哪个。我们先拍一个，设隐藏层节点数为 300 吧。

对于 3 层  $784 * 300 * 10$  的全连接网络，总共有  $300 * (784 + 1) + 10 * (300 + 1) = 238510$  个参数！神经网络之所以强大，是它提供了一种非常简单的方法去实现大量的参数。目前百亿参数、千亿样本的超大规模神经网络也是有的。因为 MNIST 只有 6 万个训练样本，参数太多了很容易过拟合，效果反而不好。

### 3.7.2 模型的训练和评估

MNIST 数据集包含 10000 个测试样本。我们先用 60000 个训练样本训练我们的网络，然后再用测试样本对网络进行测试，计算识别错误率：

$$\text{错误率} = \frac{\text{错误预测样本数}}{\text{总样本数}}$$

我们每训练 10 轮，评估一次准确率。当准确率开始下降时（出现了过拟合）终止训练。

### 3.7.3 代码实现

首先，我们需要把 MNIST 数据集处理为神经网络能够接受的形式。MNIST 训练集的文件格式可以参考官方网站，这里不在赘述。每个训练样本是一个 28\*28 的图像，我们按照行优先，把它转化为一个 784 维的向量。每个标签是 0-9 的值，我们将其转换为一个 10 维的 one-hot 向量：如果标签值为  $n$ ，我们就把向量的第  $n$  维（从 0 开始编号）设置为 0.9，而其它维设置为 0.1。例如，向量 [0.1, 0.1, 0.9, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1] 表示值 2。

下面是处理 MNIST 数据的代码：

```
1 #!/usr/bin/env python
2 # -*- coding: UTF-8 -*-
```



```
3 import struct
4 from bp import *
5 from datetime import datetime
6 # 数据加载器基类
7 class Loader(object):
8     def __init__(self, path, count):
9         ...
10    初始化加载器
11    path: 数据文件路径
12    count: 文件中的样本个数
13    ...
14    self.path = path
15    self.count = count
16    def get_file_content(self):
17        ...
18    读取文件内容
19    ...
20    f = open(self.path, 'rb')
21    content = f.read()
22    f.close()
23    return content
24    def to_int(self, byte):
25        ...
26        将 unsigned byte 字符转换为整数
27        ...
28        return struct.unpack('B', byte)[0]
29 # 图像数据加载器
30 class ImageLoader(Loader):
31     def get_picture(self, content, index):
32         ...
33         内部函数，从文件中获取图像
34         ...
35         start = index * 28 * 28 + 16
36         picture = []
37         for i in range(28):
38             picture.append([])
39             for j in range(28):
40                 picture[i].append(
41                     self.to_int(content[start + i * 28 + j]))
42         return picture
43     def get_one_sample(self, picture):
44         ...
45         内部函数，将图像转化为样本的输入向量
```

```
46     ...
47     sample = []
48     for i in range(28):
49         for j in range(28):
50             sample.append(picture[i][j])
51     return sample
52 def load(self):
53     ...
54     加载数据文件，获得全部样本的输入向量
55     ...
56     content = self.get_file_content()
57     data_set = []
58     for index in range(self.count):
59         data_set.append(
60             self.get_one_sample(
61                 self.get_picture(content, index)))
62     return data_set
63 # 标签数据加载器
64 class LabelLoader(Loader):
65     def load(self):
66         ...
67         加载数据文件，获得全部样本的标签向量
68         ...
69         content = self.get_file_content()
70         labels = []
71         for index in range(self.count):
72             labels.append(self.norm(content[index + 8]))
73         return labels
74     def norm(self, label):
75         ...
76         内部函数，将一个值转换为10维标签向量
77         ...
78         label_vec = []
79         label_value = self.to_int(label)
80         for i in range(10):
81             if i == label_value:
82                 label_vec.append(0.9)
83             else:
84                 label_vec.append(0.1)
85         return label_vec
86     def get_training_data_set():
87         ...
88     获得训练数据集
```

```

89     '',
90     image_loader = ImageLoader('train-images-idx3-ubyte', 60000)
91     label_loader = LabelLoader('train-labels-idx1-ubyte', 60000)
92     return image_loader.load(), label_loader.load()
93 def get_test_data_set():
94     '',
95     获得测试数据集
96     '',
97     image_loader = ImageLoader('t10k-images-idx3-ubyte', 10000)
98     label_loader = LabelLoader('t10k-labels-idx1-ubyte', 10000)
99     return image_loader.load(), label_loader.load()

```

网络的输出是一个 10 维向量，这个向量第  $n$  个(从 0 开始编号)元素的值最大，那么  $n$  就是网络的识别结果。下面是代码实现：

```

1 def get_result(vec):
2     max_value_index = 0
3     max_value = 0
4     for i in range(len(vec)):
5         if vec[i] > max_value:
6             max_value = vec[i]
7             max_value_index = i
8     return max_value_index

```

我们使用错误率来对网络进行评估，下面是代码实现：

```

1 def evaluate(network, test_data_set, test_labels):
2     error = 0
3     total = len(test_data_set)
4     for i in range(total):
5         label = get_result(test_labels[i])
6         predict = get_result(network.predict(test_data_set[i]))
7         if label != predict:
8             error += 1
9     return float(error) / float(total)

```

最后实现我们的训练策略：每训练 10 轮，评估一次准确率，当准确率开始下降时终止训练。下面是代码实现：

```

1 def train_and_evaluate():
2     last_error_ratio = 1.0
3     epoch = 0
4     train_data_set, train_labels = get_training_data_set()
5     test_data_set, test_labels = get_test_data_set()
6     network = Network([784, 300, 10])
7     while True:
8         epoch += 1

```

```

9     network.train(train_labels, train_data_set, 0.3, 1)
10    print '%s epoch %d finished' % (now(), epoch)
11    if epoch % 10 == 0:
12        error_ratio = evaluate(network, test_data_set,
13                                test_labels)
14        print '%s after epoch %d, error ratio is %f' % (now(),
15                                                          epoch, error_ratio)
16        if error_ratio > last_error_ratio:
17            break
18    else:
19        last_error_ratio = error_ratio
18 if __name__ == '__main__':
19     train_and_evaluate()

```

在我的机器上测试了一下，1个 epoch 大约需要 9000 多秒，所以要对代码做很多的性能优化工作（比如用向量化编程）。训练要很久很久，可以把它上传到服务器上，在 tmux 的 session 里面去运行。为了防止异常终止导致前功尽弃，我们每训练 10 轮，就把获得参数值保存在磁盘上，以便后续可以恢复。（代码略）

### 3.7.4 向量化编程

 **注意** 完整代码请参考 GitHub:[https://github.com/hanbt/learn\\_dl/blob/master/fc.py](https://github.com/hanbt/learn_dl/blob/master/fc.py) (python2.7)

在经历了漫长的训练之后，我们可能会想到，肯定有更好的办法！是的，程序员们，现在我们需要告别面向对象编程了，转而去使用另外一种更适合深度学习算法的编程方式：向量化编程。主要有两个原因：一个是我们事实上并不需要真的去定义 Node、Connection 这样的对象，直接把数学计算实现了就可以了；另一个原因，是底层算法库会针对向量运算做优化（甚至有专用的硬件，比如 GPU），程序效率会提升很多。所以，在深度学习的世界里，我们总会想法设法的把计算表达为向量的形式。我相信优秀的程序员不会把自己拘泥于某种（自己熟悉的）编程范式上，而会去学习并使用最为合适的范式。

下面，我们用向量化编程的方法，重新实现前面的全连接神经网络。

首先，我们需要把所有的计算都表达为向量的形式。对于全连接神经网络来说，主要有三个计算公式。

前向计算，我们发现公式3.2已经是向量化的表达了：

$$\vec{a} = \sigma(W \cdot \vec{x}) \quad (3.7)$$

上式中的  $\sigma$  表示 sigmoid 函数。

反向计算，我们需要把公式3.3和公式3.4使用向量来表示：

$$\vec{\delta} = \vec{y}(1 - \vec{y})(\vec{t} - \vec{y}) \quad (3.8)$$

$$\vec{\delta}^{(l)} = \vec{a}^{(l)}(1 - \vec{a}^{(l)})W^T \vec{\delta}^{(l+1)} \quad (3.9)$$

在公式3.9中， $\vec{\delta}^{(l)}$  表示第 l 层的误差项； $W^T$  表示矩阵  $W$  的转置。

我们还需要权重数组  $\mathbf{W}$  和偏置项  $\mathbf{b}$  的梯度计算的向量化表示。也就是需要把公式3.5使用向量化表示：

$$w_{ji} \leftarrow w_{ji} + \eta \delta_j x_{ji} \quad (3.10)$$

其对应的向量化表示为：

$$\mathbf{W} \leftarrow \mathbf{W} + \eta \vec{\delta} \vec{x}^T \quad (3.11)$$

更新偏置项的向量化表示为：

$$\vec{b} \leftarrow \vec{b} + \eta \vec{\delta} \quad (3.12)$$

现在，我们根据上面几个公式，重新实现一个类：FullConnectedLayer。它实现了全连接层的前向和后向计算：

```

1 # 全连接层实现类
2 class FullConnectedLayer(object):
3     def __init__(self, input_size, output_size,
4                  activator):
5         ...
6         构造函数
7         input_size: 本层输入向量的维度
8         output_size: 本层输出向量的维度
9         activator: 激活函数
10        ...
11        self.input_size = input_size
12        self.output_size = output_size
13        self.activator = activator
14        # 权重数组W
15        self.W = np.random.uniform(-0.1, 0.1,
16                                   (output_size, input_size))
17        # 偏置项b
18        self.b = np.zeros((output_size, 1))
19        # 输出向量
20        self.output = np.zeros((output_size, 1))
21    def forward(self, input_array):
22        ...
23        前向计算
24        input_array: 输入向量, 维度必须等于 input_size
25        ...
26        # 式2
27        self.input = input_array
28        self.output = self.activator.forward(
29            np.dot(self.W, input_array) + self.b)
30    def backward(self, delta_array):
31        ...

```

```
32     反向计算W和b的梯度
33     delta_array: 从上一层传递过来的误差项
34     ...
35     # 式8
36     self.delta = self.activator.backward(self.input) * np.dot(
37         self.W.T, delta_array)
38     self.W_grad = np.dot(delta_array, self.input.T)
39     self.b_grad = delta_array
40     def update(self, learning_rate):
41         ...
42         使用梯度下降算法更新权重
43         ...
44         self.W += learning_rate * self.W_grad
45         self.b += learning_rate * self.b_grad
```

上面这个类一举取代了原先的 Layer、Node、Connection 等类，不但代码更加容易理解，而且运行速度也快了几百倍。

现在，我们对 Network 类稍作修改，使之用到 FullConnectedLayer：

```
1 # Sigmoid 激活函数类
2 class SigmoidActivator(object):
3     def forward(self, weighted_input):
4         return 1.0 / (1.0 + np.exp(-weighted_input))
5     def backward(self, output):
6         return output * (1 - output)
7 # 神经网络类
8 class Network(object):
9     def __init__(self, layers):
10         ...
11         构造函数
12         ...
13         self.layers = []
14         for i in range(len(layers) - 1):
15             self.layers.append(
16                 FullConnectedLayer(
17                     layers[i], layers[i+1],
18                     SigmoidActivator()
19                 )
20             )
21     def predict(self, sample):
22         ...
23         使用神经网络实现预测
24         sample: 输入样本
25         ...
```

```
26     output = sample
27     for layer in self.layers:
28         layer.forward(output)
29         output = layer.output
30     return output
31 def train(self, labels, data_set, rate, epoch):
32     ...
33     训练函数
34     labels: 样本标签
35     data_set: 输入样本
36     rate: 学习速率
37     epoch: 训练轮数
38     ...
39     for i in range(epoch):
40         for d in range(len(data_set)):
41             self.train_one_sample(labels[d],
42                                   data_set[d], rate)
43     def train_one_sample(self, label, sample, rate):
44         self.predict(sample)
45         self.calc_gradient(label)
46         self.update_weight(rate)
47     def calc_gradient(self, label):
48         delta = self.layers[-1].activator.backward(
49             self.layers[-1].output
50         ) * (label - self.layers[-1].output)
51         for layer in self.layers[::-1]:
52             layer.backward(delta)
53             delta = layer.delta
54         return delta
55     def update_weight(self, rate):
56         for layer in self.layers:
57             layer.update(rate)
```

现在，Network 类也清爽多了，用我们的新代码再次训练一下 MNIST 数据集吧。

## 3.8 小结

至此，你已经完成了又一次漫长的学习之旅。你现在应该已经明白了神经网络的基本原理，高兴的话，你甚至有能力去动手实现一个，并用它解决一些问题。如果感到困难也不要气馁，这篇文章是一个重要的分水岭，如果你完全弄明白了的话，在真正的『小白』和装腔作势的『大牛』面前吹吹牛是完全没有问题的。

作为深度学习入门的系列文章，本文也是上半场的结束。在这个半场，你掌握了机器学习、神经网络的基本概念，并且有能力去动手解决一些简单的问题（例如手写数字

识别，如果用传统的观点来看，其实这些问题也不简单）。而且，一旦掌握基本概念，后面的学习就容易多了。

在下半场，我们讲介绍更多『深度』学习的内容，我们已经讲了神经网络 (Neutrol Network)，但是并没有讲深度神经网络 (Deep Neutrol Network)。Deep 会带来更加强大的能力，同时也带来更多问题。如果不理解这些问题和它们的解决方案，也不能说你入门了『深度』学习。

目前业界有很多开源的神经网络实现，它们的功能也要强大的多，因此你并不需要事必躬亲的去实现自己的神经网络。我们在上半场不断的从头发明轮子，是为了让你明白神经网络的基本原理，这样你就能非常迅速的掌握这些工具。在下半场的文章中，我们改变了策略：不会再从头开始去实现，而是尽可能应用现有的工具。

下一篇文章，我们介绍不同结构的神经网络，比如鼎鼎大名的**卷积神经网络**，它在图像和语音领域已然创造了诸多奇迹，在自然语言处理领域的研究也如火如荼。某种意义上说，它的成功大大提升了人们对于深度学习的信心。



# 第4章 卷积神经网络

## 内容提要

- 激活函数:Relu 4.1
- 全连接网络 VS 卷积网络 4.2
- 卷积神经网络是啥 4.3
- 卷积神经网络输出值的计算 4.4
- 卷积层输出值的计算 4.4.1
- Pooling 层输出值的计算 4.4.2
- 卷积神经网络的训练 4.5
- 卷积层的训练 4.5.1
- Pooling 层的训练 4.5.2
- 编程实战：卷积神经网络的实现 4.6
- 卷积层的实现 4.6.1
- Max Pooling 层的实现 4.6.2
- 卷积神经网络的应用 4.7

在前面的文章中，我们介绍了全连接神经网络，以及它的训练和使用。我们用它来识别了手写数字，然而，这种结构的网络对于图像识别任务来说并不是很合适。本文将要介绍一种更适合图像、语音识别任务的神经网络结构——卷积神经网络 (Convolutional Neural Network, CNN)。说卷积神经网络是最重要的一种神经网络也不为过，它在最近几年大放异彩，几乎所有图像、语音识别领域的重要突破都是卷积神经网络取得的，比如谷歌的 GoogleNet、微软的 ResNet 等，打败李世石的 AlphaGo 也用到了这种网络。本文将详细介绍卷积神经网络以及它的训练算法，以及动手实现一个简单的卷积神经网络。

## 4.1 激活函数:Relu

最近几年卷积神经网络中，激活函数往往不选择 sigmoid 或 tanh 函数，而是选择 relu 函数。Relu 函数的定义是：

$$f(x) = \max(0, x)$$

Relu 函数图像如图4.1所示：

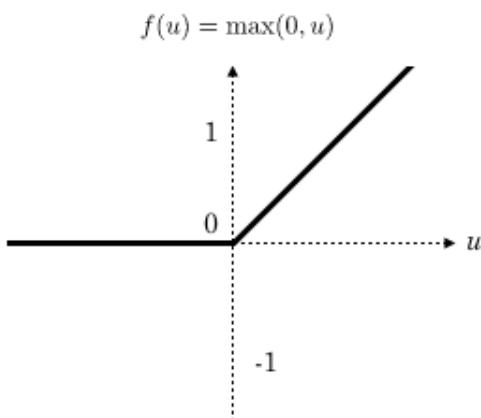


图 4.1: Relu 函数

Relu 函数作为激活函数，有下面几大优势：

- **速度快**和 sigmoid 函数需要计算指数和倒数相比，relu 函数其实就是一个  $\max(0,x)$ ，计算代价小很多。
- **减轻梯度消失问题**回忆一下计算梯度的公式  $\nabla = \sigma' \delta x$ 。其中， $\sigma'$  是 sigmoid 函数的导数。在使用反向传播算法进行梯度计算时，每经过一层 sigmoid 神经元，梯度就要乘上一个  $\sigma'$ 。从下图可以看出， $\sigma'$  函数最大值是  $1/4$ 。因此，乘一个  $\sigma'$  会导致梯度越来越小，这对于深层网络的训练是个很大的问题。而 relu 函数的导数是 1，不会导致梯度变小。当然，激活函数仅仅是导致梯度减小的一个因素，但无论如何在这方面 relu 的表现强于 sigmoid。使用 relu 激活函数可以让你训练更深的网络。
- **稀疏性**通过对大脑的研究发现，大脑在工作的时候只有大约 5% 的神经元是激活的，而采用 sigmoid 激活函数的人工神经网络，其激活率大约是 50%。有论文声称人工神经网络在 15%-30% 的激活率时是比较理想的。因为 relu 函数在输入小于 0 时是完全不激活的，因此可以获得一个更低的激活率。

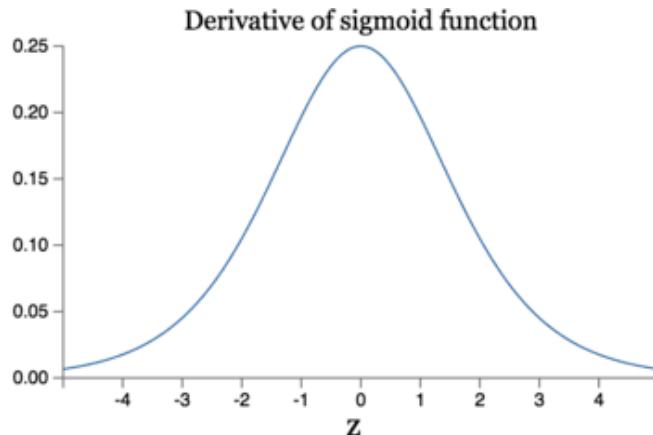


图 4.2: Relu 函数导数

## 4.2 全连接网络 VS 卷积网络

全连接神经网络之所以不太适合图像识别任务，主要有以下几个方面的问题：

- **参数数量太多**考虑一个输入  $1000*1000$  像素的图片 (一百万像素，现在已经不能算大图了)，输入层有  $1000*1000=100$  万节点。假设第一个隐藏层有 100 个节点 (这个数量并不多)，那么仅这一层就有  $(1000*1000+1)*100=1$  亿参数，这实在是太多了！我们看到图像只扩大一点，参数数量就会多很多，因此它的扩展性很差。
- **没有利用像素之间的位置信息**对于图像识别任务来说，每个像素和其周围像素的联系是比较紧密的，和离得很远的像素的联系可能就很小了。如果一个神经元和上一层所有神经元相连，那么就相当于对于一个像素来说，把图像的所有像素都等同看待，这不符合前面的假设。当我们完成每个连接权重的学习之后，最终可能会发现，有大量的权重，它们的值都是很小的 (也就是这些连接其实无关紧要)。努力学习大量并不重要的权重，这样的学习必将是非常低效的。

• **网络层数限制**我们知道网络层数越多其表达能力越强,但是通过梯度下降方法训练深度全连接神经网络很困难,因为全连接神经网络的梯度很难传递超过3层。因此,我们不可能得到一个很深的全连接神经网络,也就限制了它的能力。

那么,卷积神经网络又是怎样解决这个问题的呢?主要有三个思路:

- **局部连接**这个是最容易想到的,每个神经元不再和上一层的所有神经元相连,而只和一小部分神经元相连。这样就减少了很多参数。
- **权值共享**一组连接可以共享同一个权重,而不是每个连接有一个不同的权重,这样又减少了很多参数。
- **下采样**可以使用 Pooling 来减少每层的样本数,进一步减少参数数量,同时还可以提升模型的鲁棒性。

对于图像识别任务来说,卷积神经网络通过尽可能保留重要的参数,去掉大量不重要的参数,来达到更好的学习效果。

接下来,我们将详述卷积神经网络到底是何方神圣。

### 4.3 卷积神经网络是啥

首先,我们先获取一个感性认识,图4.3是一个卷积神经网络的示意图:

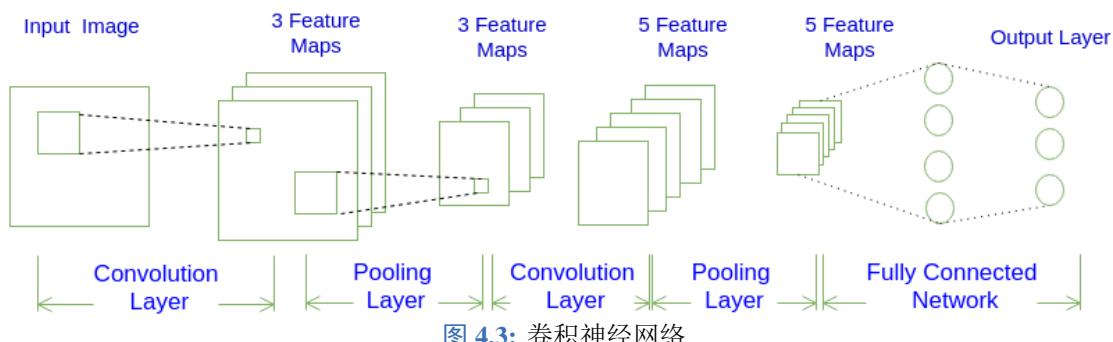


图 4.3: 卷积神经网络

#### 4.3.1 网络架构

如图4.3所示,一个卷积神经网络由若干卷积层、Pooling 层、全连接层组成。你可以构建各种不同的卷积神经网络,它的常用架构模式为:

```
INPUT -> [[CONV]*N -> POOL?] *M -> [FC]*K
```

也就是 N 个卷积层叠加,然后(可选)叠加一个 Pooling 层,重复这个结构 M 次,最后叠加 K 个全连接层。

对于图4.3展示的卷积神经网络:

```
INPUT -> CONV -> POOL -> CONV -> POOL -> FC -> FC
```

按照上述模式可以表示为:

```
INPUT -> [[CONV]*1 -> POOL]*2 -> [FC]*2
```

也就是: N=1, M=2, K=2。

### 4.3.2 三维的层结构

从图4.3我们可以发现卷积神经网络的层结构和全连接神经网络的层结构有很大不同。全连接神经网络每层的神经元是按照一维排列的，也就是排成一条线的样子；而卷积神经网络每层的神经元是按照三维排列的，也就是排成一个长方体的样子，有宽度、高度和深度。

对于图4.3展示的神经网络，我们看到输入层的宽度和高度对应于输入图像的宽度和高度，而它的深度为1。接着，第一个卷积层对这幅图像进行了卷积操作（后面我们会讲如何计算卷积），得到了三个Feature Map。这里的“3”可能是让很多初学者迷惑的地方，实际上，就是这个卷积层包含三个Filter，也就是三套参数，每个Filter都可以把原始输入图像卷积得到一个Feature Map，三个Filter就可以得到三个Feature Map。至于一个卷积层可以有多少个Filter，那是可以自由设定的。也就是说，卷积层的Filter个数也是一个超参数。我们可以把Feature Map看做是通过卷积变换提取到的图像特征，三个Filter就对原始图像提取出三组不同的特征，也就是得到了三个Feature Map，也称做三个通道（channel）。

继续观察图4.3，在第一个卷积层之后，Pooling层对三个Feature Map做了下采样（后面我们会讲如何计算下采样），得到了三个更小的Feature Map。接着，是第二个卷积层，它有5个Filter。每个Filter都把前面下采样之后的3个Feature Map卷积在一起，得到一个新的Feature Map。这样，5个Filter就得到了5个Feature Map。接着，是第二个Pooling，继续对5个Feature Map进行下采样，得到了5个更小的Feature Map。

图4.3所示网络的最后两层是全连接层。第一个全连接层的每个神经元，和上一层5个Feature Map中的每个神经元相连，第二个全连接层（也就是输出层）的每个神经元，则和第一个全连接层的每个神经元相连，这样得到了整个网络的输出。

至此，我们对卷积神经网络有了最基本的感性认识。接下来，我们将介绍卷积神经网络中各种层的计算和训练。

## 4.4 卷积神经网络输出值的计算

### 4.4.1 卷积层输出值的计算

我们用一个简单的例子来讲述如何计算卷积，然后，我们抽象出卷积层的一些重要概念和计算方法。

假设有一个 $5 \times 5$ 的图像，使用一个 $3 \times 3$ 的filter进行卷积，想得到一个 $3 \times 3$ 的Feature Map，如图4.4所示。为了清楚的描述卷积计算过程，我们首先对图像的每个像素进行编号，用 $x_{i,j}$ 表示图像的第*i*行第*j*列元素；对filter的每个权重进行编号，用 $w_{m,n}$ 表示第*m*行第*n*列权重，用 $w_b$ 表示filter的偏置项；对Feature Map的每个元素进行编号，用 $a_{i,j}$ 表示Feature Map的第*i*行第*j*列元素；用 $f$ 表示激活函数（这个例子选择relu函数作为激活函数）。然后，使用下列公式计算卷积：

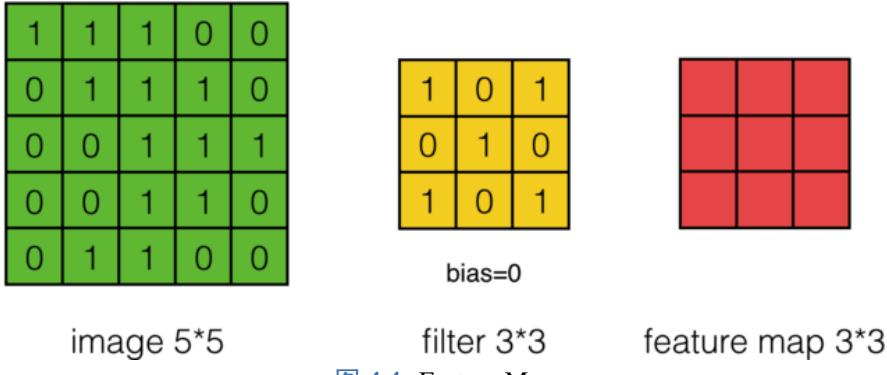


图 4.4: Feature Map

$$a_{i,j} = f\left(\sum_{m=0}^2 \sum_{n=0}^2 w_{m,n} x_{i+m, j+n} + w_b\right) \quad (4.1)$$

例如，对于 Feature Map 左上角元素  $a_{0,0}$  来说，其卷积计算方法为：

$$\begin{aligned} a_{0,0} &= f\left(\sum_{m=0}^2 \sum_{n=0}^2 w_{m,n} x_{m+0, n+0} + w_b\right) \\ &= \text{relu}(w_{0,0}x_{0,0} + w_{0,1}x_{0,1} + w_{0,2}x_{0,2} + w_{1,0}x_{1,0} + w_{1,1}x_{1,1} \\ &\quad + w_{1,2}x_{1,2} + w_{2,0}x_{2,0} + w_{2,1}x_{2,1} + w_{2,2}x_{2,2} + w_b) \\ &= \text{relu}(1 + 0 + 1 + 0 + 1 + 0 + 0 + 0 + 1 + 0) = \text{relu}(4) = 4 \end{aligned}$$

计算结果如图4.5所示：

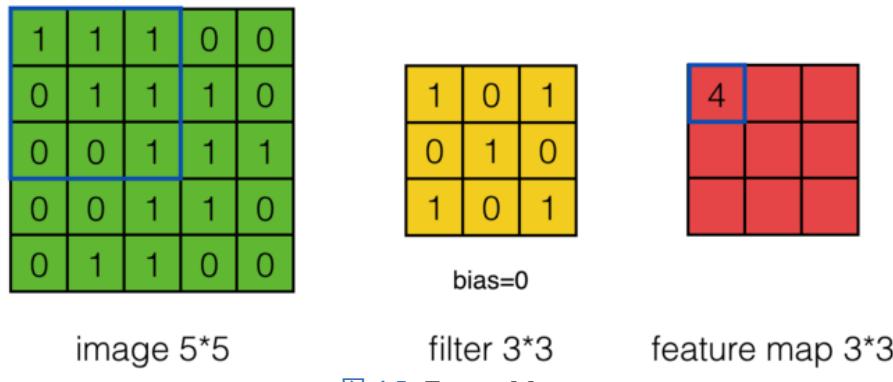


图 4.5: Feature Map

接下来，Feature Map 的元素  $a_{0,1}$  的卷积计算方法为：

$$\begin{aligned} a_{0,1} &= f\left(\sum_{m=0}^2 \sum_{n=0}^2 w_{m,n} x_{m+0, n+1} + w_b\right) \\ &= \text{relu}(w_{0,0}x_{0,1} + w_{0,1}x_{0,2} + w_{0,2}x_{0,3} + w_{1,0}x_{1,1} + w_{1,1}x_{1,2} \\ &\quad + w_{1,2}x_{1,3} + w_{2,0}x_{2,1} + w_{2,1}x_{2,2} + w_{2,2}x_{2,3} + w_b) \\ &= \text{relu}(1 + 0 + 0 + 0 + 1 + 0 + 0 + 0 + 1 + 0) \\ &= \text{relu}(3) = 3 \end{aligned}$$

计算结果如图4.6所示：

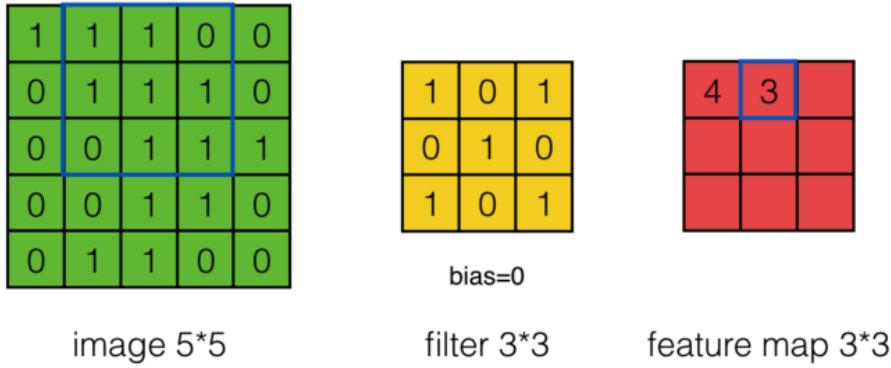


图 4.6: Feature Map

可以依次计算出 Feature Map 中所有元素的值。图4.7显示了整个 Feature Map 的计算过程：

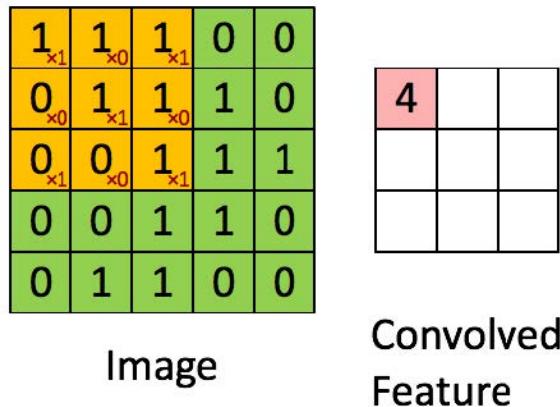


图 4.7: Feature Map

上面的计算过程中，步幅 (stride) 为 1。步幅可以设为大于 1 的数。例如，当步幅为 2 时，Feature Map 计算如图4.8。

我们注意到，当步幅设置为 2 的时候，Feature Map 就变成 2\*2 了。这说明图像大小、步幅和卷积后的 Feature Map 大小是有关系的。事实上，它们满足下面的关系：

$$W_2 = (W_1 - F + 2P)/S + 1 \quad (4.2)$$

$$H_2 = (H_1 - F + 2P)/S + 1 \quad (4.3)$$

在上面两个公式中， $W_2$  是卷积后 Feature Map 的宽度； $W_1$  是卷积前图像的宽度； $F$  是 filter 的宽度； $P$  是 **Zero Padding** 数量，**Zero Padding** 是指在原始图像周围补几圈 0，如果  $P$  的值是 1，那么就补 1 圈 0； $S$  是步幅； $H_2$  是卷积后 Feature Map 的高度； $H_1$  是卷积前图像的高度。公式4.2和4.3本质上是一样的。

以前面的例子来说，图像宽度  $W_1 = 5$ ，filter 宽度  $F = 3$ ，**Zero Padding**  $P = 0$ ，步幅  $S = 2$ ，则

$$W_2 = (W_1 - F + 2P)/S + 1 = (5 - 3 + 0)/2 + 1 = 2$$

说明 Feature Map 宽度是 2。同样，我们也可以计算出 Feature Map 高度也是 2。

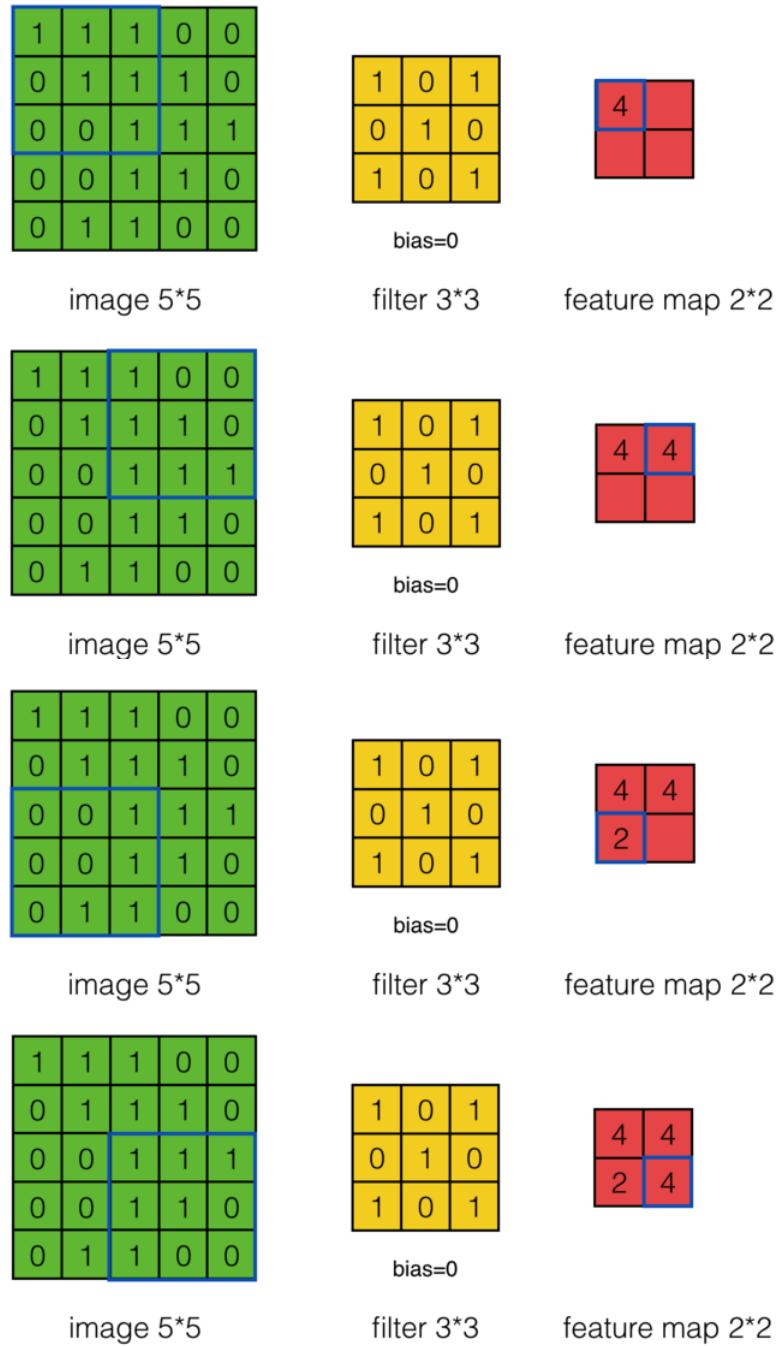


图 4.8: Feature Map 计算过程

前面我们已经讲了深度为 1 的卷积层的计算方法，如果深度大于 1 怎么计算呢？其实也是类似的。如果卷积前的图像深度为 D，那么相应的 filter 的深度也必须为 D。我们扩展一下公式4.1，得到了深度大于 1 的卷积计算公式：

$$a_{i,j} = f\left(\sum_{d=0}^{D-1} \sum_{m=0}^{F-1} \sum_{n=0}^{F-1} w_{d,m,n} x_{d,i+m,j+n} + w_b\right) \quad (4.4)$$

在公式4.4中，D 是深度；F 是 filter 的大小（宽度或高度，两者相同）； $w_{d,m,n}$  表示 filter 的第 d 层第 m 行第 n 列权重； $a_{d,i,j}$  表示图像的第 d 层第 i 行第 j 列像素；其它的符号含义和公式4.1是相同的，不再赘述。

我们前面还曾提到，每个卷积层可以有多个 filter。每个 filter 和原始图像进行卷积后，都可以得到一个 Feature Map。因此，卷积后 Feature Map 的深度（个数）和卷积层的 filter 个数是相同的。

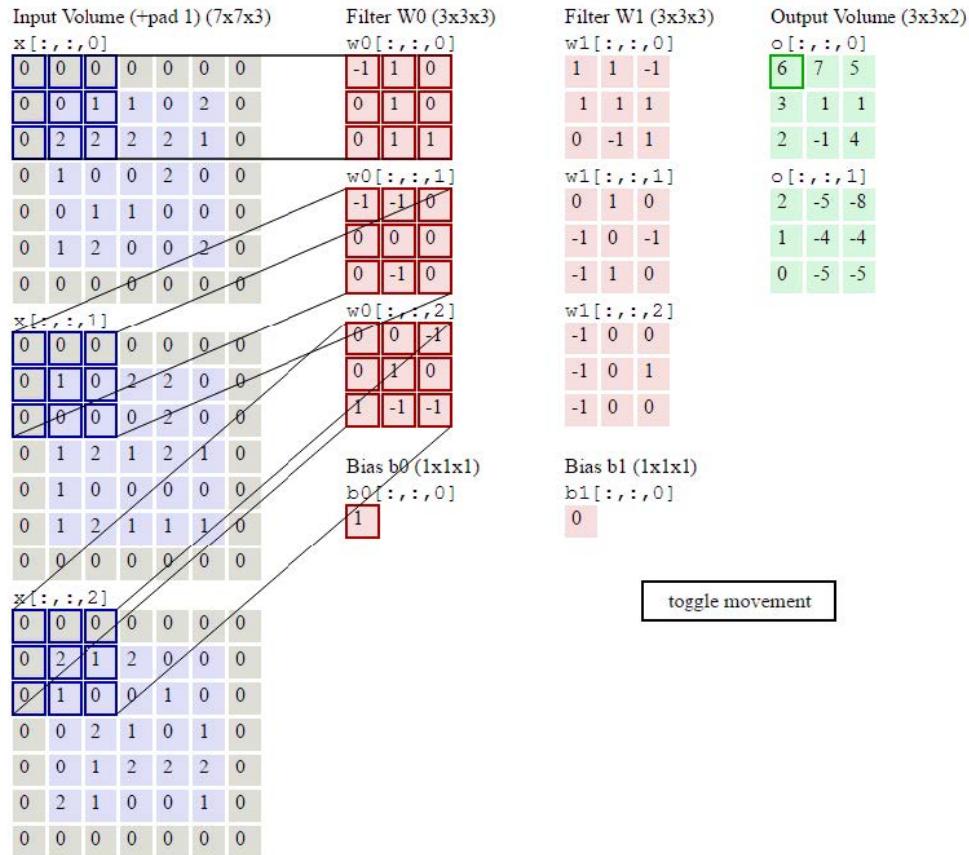


图 4.9: 卷积过程

图4.9显示了包含两个 filter 的卷积层的计算。我们可以看到  $7 \times 7 \times 3$  输入，经过两个  $3 \times 3 \times 3$  filter 的卷积（步幅为 2），得到了  $3 \times 3 \times 2$  的输出。另外我们也会看到下图的 **Zero padding** 是 1，也就是在输入元素的周围补了一圈 0。**Zero padding** 对于图像边缘部分的特征提取是很有帮助的。

以上就是卷积层的计算方法。这里面体现了**局部连接**和**权值共享**：每层神经元只和上一层部分神经元相连（卷积计算规则），且 filter 的权值对于上一层所有神经元都是一样的。对于包含两个  $3 \times 3 \times 3$  的 filter 的卷积层来说，其参数数量仅有  $(3 \times 3 \times 3 + 1) \times 2 = 56$  个，且参数数量与上一层神经元个数无关。与全连接神经网络相比，其参数数量大大减少了。

### 用卷积公式来表达卷积层计算

不想了解太多数学细节的读者可以跳过这一节，不影响对全文的理解。

公式4.4的表达很是繁冗，最好能简化一下。就像利用矩阵可以简化表达全连接神经网络的计算一样，我们利用卷积公式可以简化卷积神经网络的表达。

下面我们介绍**二维卷积公式**。

设矩阵  $A, B$ , 其行、列数分别为  $m_a, n_a, m_b, n_b$ , 则二维卷积公式如下:

$$C_{s,t} = \sum_0^{m_a-1} \sum_0^{n_a-1} A_{m,n} B_{s-m,t-n}$$

且  $s, t$  满足条件  $0 \leq s < m_a + m_b - 1, 0 \leq t < n_a + n_b - 1$ 。

我们可以把上式写成

$$C = A * B \quad (4.5)$$

如果我们按照公式4.5来计算卷积, 我们可以发现矩阵 A 实际上是 filter, 而矩阵 B 是待卷积的输入, 位置关系也有所不同如图4.10:

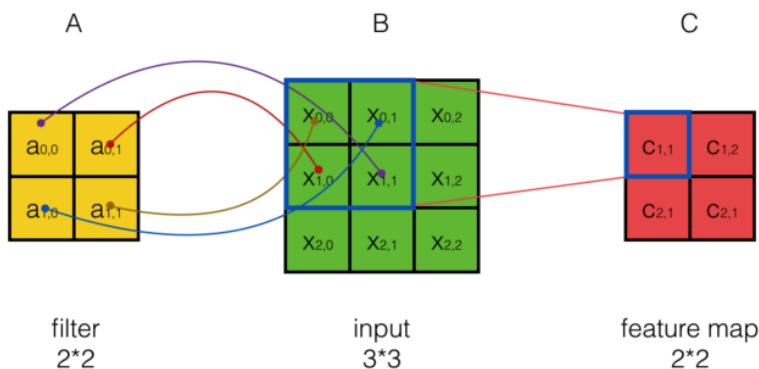


图 4.10: 位置关系图

从上图可以看到, A 左上角的值  $a_{0,0}$  与 B 对应区块中右下角的值  $b_{1,1}$  相乘, 而不是与左上角的  $b_{0,0}$  相乘。因此, 数学中的卷积和卷积神经网络中的『卷积』还是有区别的, 为了避免混淆, 我们把卷积神经网络中的『卷积』操作叫做互相关 (cross-correlation) 操作。

卷积和互相关操作是可以转化的。首先, 我们把矩阵 A 翻转 180 度, 然后再交换 A 和 B 的位置 (即把 B 放在左边而把 A 放在右边。卷积满足交换率, 这个操作不会导致结果变化), 那么卷积就变成了互相关。

如果我们不去考虑两者这么一点点的区别, 我们可以把公式4.5代入到公式4.4:

$$A = f\left(\sum_{d=0}^{D-1} X_d * W_d + w_b\right) \quad (4.6)$$

其中, A 是卷积层输出的 feature map。同公式4.4相比, 公式4.6就简单多了。然而, 这种简洁写法只适合步长为 1 的情况。

#### 4.4.2 Pooling 层输出值的计算

Pooling 层主要的作用是下采样, 通过去掉 Feature Map 中不重要的样本, 进一步减少参数数量。Pooling 的方法很多, 最常用的是 **Max Pooling**。**Max Pooling** 实际上就是在  $n*n$  的样本中取最大值, 作为采样后的样本值。图4.11是  $2*2$  max pooling。

除了 **Max Pooling** 之外, 常用的还有 **Mean Pooling**---取各样本的平均值。

对于深度为 D 的 Feature Map, 各层独立做 Pooling, 因此 Pooling 后的深度仍然为 D。

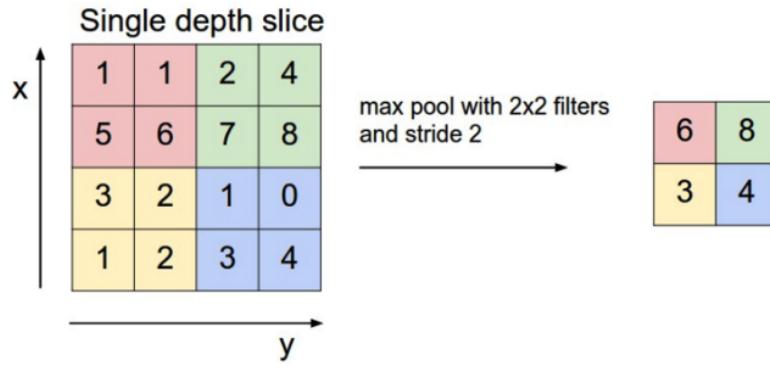


图 4.11: max pooling

#### 4.4.3 全连接层

全连接层输出值的计算和第3章神经网络和反向传播算法讲过的全连接神经网络是一样的，这里就不再赘述了。

### 4.5 卷积神经网络的训练

和全连接神经网络相比，卷积神经网络的训练要复杂一些。但训练的原理是一样的：利用链式求导计算损失函数对每个权重的偏导数（梯度），然后根据梯度下降公式更新权重。训练算法依然是反向传播算法。

我们先回忆一下第3章神经网络和反向传播算法介绍的反向传播算法，整个算法分为三个步骤：

1. 前向计算每个神经元的输出值  $a_j$  ( $j$  表示网络的第  $j$  个神经元，以下同)；
2. 反向计算每个神经元的误差项  $\delta_j$ ， $\delta_j$  在有的文献中也叫做敏感度 (sensitivity)。它实际上是网络的损失函数  $E_d$  对神经元加权输入  $net_j$  的偏导数，即  $\delta_j = \frac{\partial E_d}{\partial net_j}$ ；
3. 计算每个神经元连接权重  $w_{ji}$  的梯度 ( $w_{ji}$  表示从神经元  $i$  连接到神经元  $j$  的权重)，公式为  $\frac{\partial E_d}{\partial w_{ji}} = a_i \delta_j$ ，其中， $a_i$  表示神经元  $i$  的输出。

最后，根据梯度下降法则更新每个权重即可。

对于卷积神经网络，由于涉及到局部连接、下采样的等操作，影响到了第二步误差项  $\delta$  的具体计算方法，而权值共享影响了第三步权重  $w$  的梯度的计算方法。接下来，我们分别介绍卷积层和 Pooling 层的训练算法。

#### 4.5.1 卷积层的训练

对于卷积层，我们先来看看上面的第二步，即如何将误差项  $\delta$  传递到上一层；然后再来看看第三步，即如何计算 filter 每个权值  $w$  的梯度。

##### 卷积层误差项的传递

##### 最简单情况下误差项的传递

我们先来考虑步长为 1、输入的深度为 1、filter 个数为 1 的最简单的情况。

假设输入的大小为  $3 \times 3$ ，filter 大小为  $2 \times 2$ ，按步长为 1 卷积，我们将得到  $2 \times 2$  的 feature

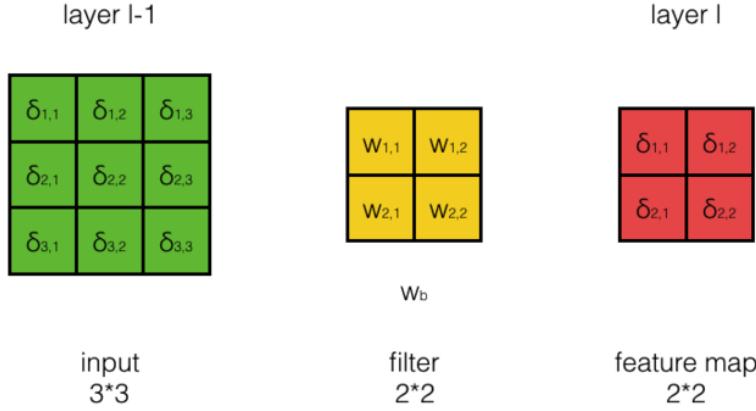


图 4.12: feature map

**map。**在图4.12中，为了描述方便，我们为每个元素都进行了编号。用  $\delta_{i,j}^{l-1}$  表示第  $l-1$  层第  $j$  行第  $j$  列的误差项；用  $w_{m,n}$  表示 filter 第  $m$  行第  $n$  列权重，用  $w_b$  表示 filter 的偏置项；用  $a_{i,j}^{l-1}$  表示第  $l-1$  层第  $i$  行第  $j$  列神经元的输出；用  $net_{i,j}^{l-1}$  表示第  $l-1$  行神经元的加权输入；用  $\delta_{i,j}^l$  表示第  $l$  层第  $j$  行第  $j$  列的误差项；用  $f^{l-1}$  表示第  $l-1$  层的激活函数。它们之间的关系如下：

$$\begin{aligned} net^l &= conv(W^l, a^{l-1}) + w_b \\ a_{i,j}^{l-1} &= f^{l-1}(net_{i,j}^{l-1}) \end{aligned}$$

上式中， $net^l$ 、 $W^l$ 、 $a^{l-1}$  都是数组， $W^l$  是由  $w_{m,n}$  组成的数组， $conv$  表示卷积操作。

在这里，我们假设第  $l$  中的每个  $\delta^l$  值都已经算好，我们要做的是计算第  $l-1$  层每个神经元的误差项  $\delta^{l-1}$ 。

根据链式求导法则：

$$\delta_{i,j}^{l-1} = \frac{\partial E_d}{\partial net_{i,j}^{l-1}} = \frac{\partial E_d}{\partial a_{i,j}^{l-1}} \frac{\partial a_{i,j}^{l-1}}{\partial net_{i,j}^{l-1}}$$

我们先求第一项  $\frac{\partial E_d}{\partial a_{i,j}^{l-1}}$ 。我们先来看几个特例，然后从中总结出一般性的规律。

**例 4.1** 计算  $\frac{\partial E_d}{\partial a_{1,1}^{l-1}}$ ， $a_{1,1}^{l-1}$  仅与  $net_{1,1}^l$  的计算有关：

$$net_{1,1}^j = w_{1,1}a_{1,1}^{l-1} + w_{1,2}a_{1,2}^{l-1} + w_{2,1}a_{2,1}^{l-1} + w_{2,2}a_{2,2}^{l-1} + w_b$$

因此：

$$\frac{\partial E_d}{\partial a_{1,1}^{l-1}} = \frac{\partial E_d}{\partial net_{1,1}^l} \frac{\partial net_{1,1}^l}{\partial a_{1,1}^{l-1}} = \delta_{1,1}^l w_{1,1}$$

**例 4.2** 计算  $\frac{\partial E_d}{\partial a_{1,2}^{l-1}}$ ， $a_{1,2}^{l-1}$  与  $net_{1,1}^l$  和  $net_{1,2}^l$  的计算都有关：

$$net_{1,1}^j = w_{1,1}a_{1,1}^{l-1} + w_{1,2}a_{1,2}^{l-1} + w_{2,1}a_{2,1}^{l-1} + w_{2,2}a_{2,2}^{l-1} + w_b$$

$$net_{1,2}^j = w_{1,1}a_{1,2}^{l-1} + w_{1,2}a_{1,3}^{l-1} + w_{2,1}a_{2,2}^{l-1} + w_{2,2}a_{2,3}^{l-1} + w_b$$

因此：

$$\frac{\partial E_d}{\partial a_{1,2}^{l-1}} = \frac{\partial E_d}{\partial net_{1,1}^l} \frac{\partial net_{1,1}^l}{\partial a_{1,2}^{l-1}} + \frac{\partial E_d}{\partial net_{1,2}^l} \frac{\partial net_{1,2}^l}{\partial a_{1,2}^{l-1}} = \delta_{1,1}^l w_{1,2} + \delta_{1,2}^l w_{1,1}$$

**例4.3** 计算  $\frac{\partial E_d}{\partial a_{2,2}^{l-1}}$ ,  $a_{2,2}^{l-1}$  与  $net_{1,1}^l$ 、 $net_{1,2}^l$ 、 $net_{2,1}^l$  和  $net_{2,2}^l$  的计算都有关:

$$\begin{aligned} net_{1,1}^j &= w_{1,1}a_{1,1}^{l-1} + w_{1,2}a_{1,2}^{l-1} + w_{2,1}a_{2,1}^{l-1} + w_{2,2}a_{2,2}^{l-1} + w_b \\ net_{1,2}^j &= w_{1,1}a_{1,2}^{l-1} + w_{1,2}a_{1,3}^{l-1} + w_{2,1}a_{2,2}^{l-1} + w_{2,2}a_{2,3}^{l-1} + w_b \\ net_{2,1}^j &= w_{1,1}a_{2,1}^{l-1} + w_{1,2}a_{2,2}^{l-1} + w_{2,1}a_{3,1}^{l-1} + w_{2,2}a_{3,2}^{l-1} + w_b \\ net_{2,2}^j &= w_{1,1}a_{2,2}^{l-1} + w_{1,2}a_{2,3}^{l-1} + w_{2,1}a_{3,2}^{l-1} + w_{2,2}a_{3,3}^{l-1} + w_b \end{aligned}$$

因此:

$$\begin{aligned} \frac{\partial E_d}{\partial a_{2,2}^{l-1}} &= \frac{\partial E_d}{\partial net_{1,1}^l} \frac{\partial net_{1,1}^l}{\partial a_{2,2}^{l-1}} + \frac{\partial E_d}{\partial net_{1,2}^l} \frac{\partial net_{1,2}^l}{\partial a_{2,2}^{l-1}} + \frac{\partial E_d}{\partial net_{2,1}^l} \frac{\partial net_{2,1}^l}{\partial a_{2,2}^{l-1}} + \frac{\partial E_d}{\partial net_{2,2}^l} \frac{\partial net_{2,2}^l}{\partial a_{2,2}^{l-1}} \\ &= \delta_{1,1}^l w_{2,2} + \delta_{1,2}^l w_{2,1} + \delta_{2,1}^l w_{1,2} + \delta_{2,2}^l w_{1,1} \end{aligned}$$

从上面三个例子，我们发挥一下想象力，不难发现，计算  $\frac{\partial E_d}{\partial a_{i,j}^{l-1}}$ ，相当于把第  $l$  层的 sensitive map 周围补一圈 0，在与 180 度翻转后的 filter 进行 **cross-correlation**，就能得到想要结果，如图4.13所示。

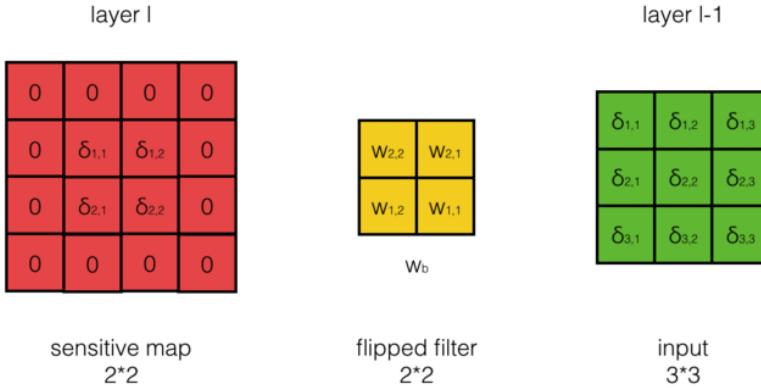


图 4.13: cross correlation

因为卷积相当于将 filter 旋转 180 度的 **cross-correlation**，因此上图的计算可以用卷积公式完美的表达:

$$\frac{\partial E_d}{\partial a_l} = \delta^l * W^l$$

上式中的  $W^l$  表示第  $l$  层的 filter 的权重数组。也可以把上式的卷积展开，写成求和的形式:

$$\frac{\partial E_d}{\partial a_{i,j}^l} = \sum_m \sum_n w_{m,n}^l \delta_{i+m, j+n}^l$$

现在，我们再求第二项  $\frac{\partial a_{i,j}^{l-1}}{\partial net_{i,j}^{l-1}}$ 。因为  $a_{i,j}^{l-1} = f(net_{i,j}^{l-1})$  所以这一项极其简单，仅求激活函数  $f$  的导数就行了。

$$\frac{\partial a_{i,j}^{l-1}}{\partial net_{i,j}^{l-1}} = f'(net_{i,j}^{l-1})$$

将第一项和第二项组合起来，我们得到最终的公式:

$$\delta_{i,j}^{l-1} = \frac{\partial E_d}{\partial net_{i,j}^{l-1}} = \frac{\partial E_d}{\partial a_{i,j}^{l-1}} \frac{\partial a_{i,j}^{l-1}}{\partial net_{i,j}^{l-1}} = \sum_m \sum_n w_{m,n}^l \delta_{i+m, j+n}^l f'(net_{i,j}^{l-1}) \quad (4.7)$$

也可以将公式4.7写成卷积的形式：

$$\delta^{l-1} = \delta^l * W^l \circ f'(net^{l-1}) \quad (4.8)$$

其中，符号 $\circ$ 表示 **element-wise product**，即将矩阵中每个对应元素相乘。注意公式4.8中的 $\delta^{l-1}$ 、 $\delta^l$ 、 $net^{l-1}$ 都是矩阵。

以上就是步长为1、输入的深度为1、filter个数为1的最简单的情况，卷积层误差项传递的算法。下面我们来推导一下步长为S的情况。

### 卷积步长为S时的误差传递

我们先来看看步长为S与步长为1的差别。

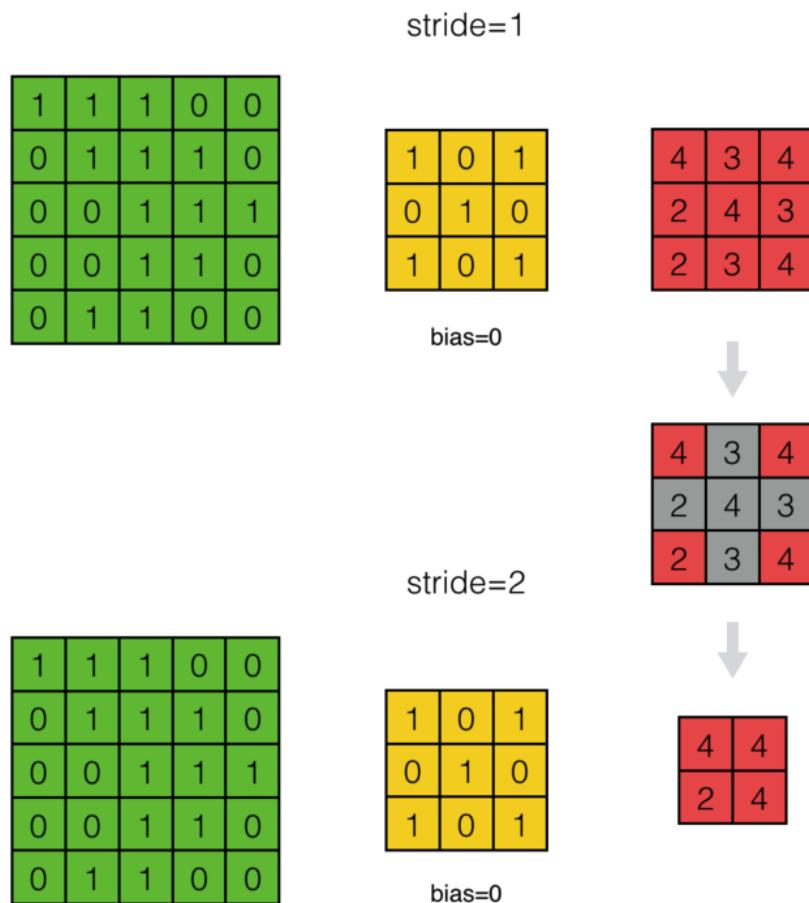


图 4.14: 卷积步长的差别

如图4.14，上面是步长为1时的卷积结果，下面是步长为2时的卷积结果。我们可以看出，因为步长为2，得到的 feature map 跳过了步长为1时相应的部分。因此，当我们反向计算误差项时，我们可以对步长为S的 sensitivity map 相应的位置进行补0，将其『还原』成步长为1时的 sensitivity map，再用公式4.8进行求解。

### 输入层深度为D时的误差传递

当输入深度为D时，filter的深度也必须为D， $l-1$ 层的 $d_i$ 通道只与filter的 $d_i$ 通道的权重进行计算。因此，反向计算误差项时，我们可以使用式8，用filter的第 $d_i$ 通道权重对第 $l$ 层 sensitivity map 进行卷积，得到第 $l-1$ 层 $d_i$ 通道的 sensitivity map。如图4.15所示：

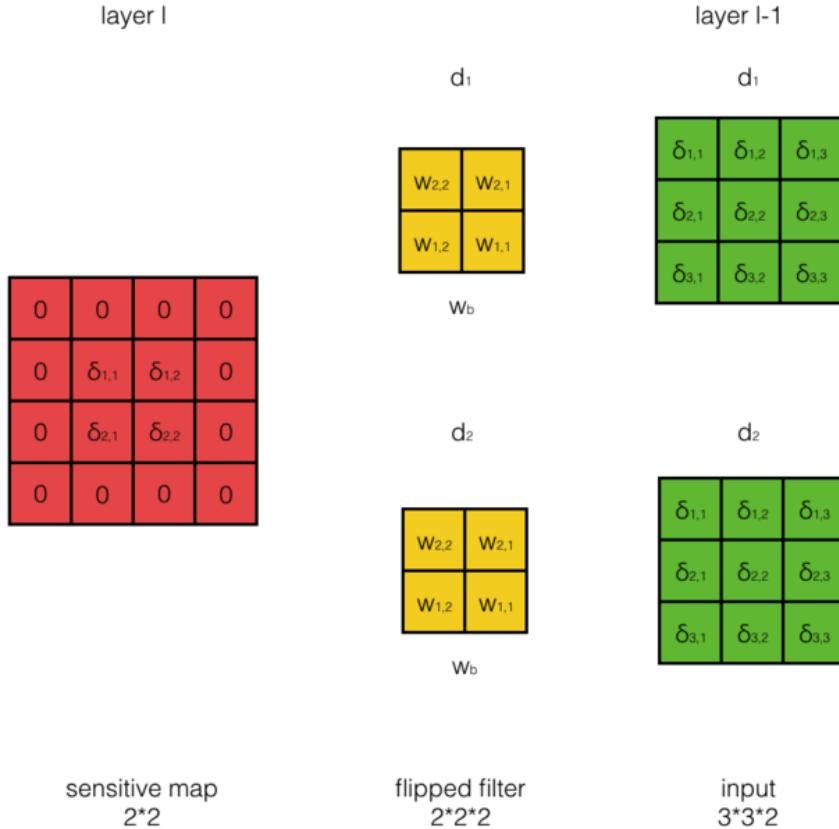


图 4.15: sensitivity map 卷积

### filter 数量为 N 时的误差传递

filter 数量为 N 时，输出层的深度也为 N，第  $i$  个 filter 卷积产生输出层的第  $i$  个 feature map。由于第  $l-1$  层每个加权输入  $net_{d,i,j}^{l-1}$  都同时影响了第  $l$  层所有 feature map 的输出值，因此，反向计算误差项时，需要使用全导数公式。也就是，我们先使用第  $d$  个 filter 对第  $l$  层相应的第  $d$  个 sensitivity map 进行卷积，得到一组 N 个  $l-1$  层的偏 sensitivity map。依次用每个 filter 做这种卷积，就得到 D 组偏 sensitivity map。最后在各组之间将 N 个偏 sensitivity map 按元素相加，得到最终的 N 个  $l-1$  层的 sensitivity map：

$$\delta^{l-1} = \sum_{d=0}^D \delta_d^l * W_d^l \circ f'(net^{l-1}) \quad (4.9)$$

以上就是卷积层误差项传递的算法，如果读者还有所困惑，可以参考后面的代码实现来理解。

### 卷积层 filter 权重梯度的计算

我们要在得到第  $l$  层 sensitivity map 的情况下，计算 filter 的权重的梯度，由于卷积层是权重共享的，因此梯度的计算稍有不同。

如图4.16所示， $a_{i,j}^l$  是第  $l-1$  层的输出， $w_{i,j}$  是第  $l$  层 filter 的权重， $\delta_{i,j}^l$  是第  $l$  层的 sensitivity map。我们的任务是计算  $w_{i,j}$  的梯度，即  $\frac{\partial E_d}{\partial w_{i,j}}$ 。

为了计算偏导数，我们需要考察权重  $w_{i,j}$  对  $E_d$  的影响。权重项  $w_{i,j}$  通过影响  $net_{i,j}^l$  的值，进而影响  $E_d$ 。我们仍然通过几个具体的例子来看权重项  $w_{i,j}$  对  $net_{i,j}^l$  的影响，然

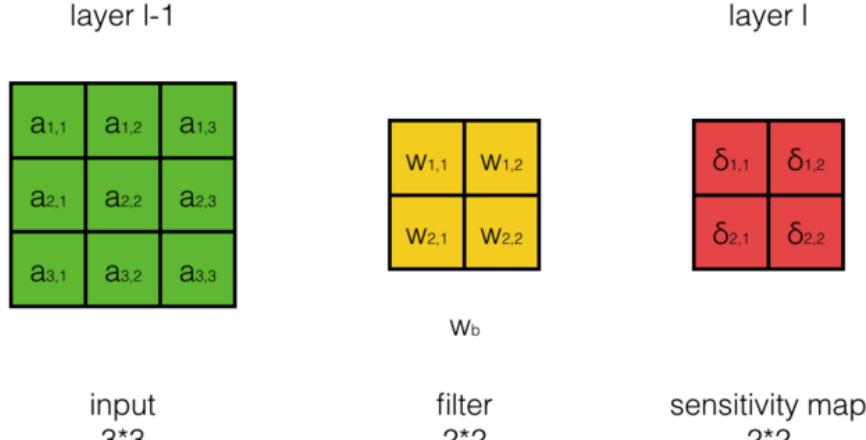


图 4.16: 梯度的计算

后再从中总结出规律。

**例 4.4** 计算  $\frac{\partial E_d}{\partial w_{1,1}}$ :

$$\begin{aligned} net_{1,1}^j &= w_{1,1}a_{1,1}^{l-1} + w_{1,2}a_{1,2}^{l-1} + w_{2,1}a_{2,1}^{l-1} + w_{2,2}a_{2,2}^{l-1} + w_b \\ net_{1,2}^j &= w_{1,1}a_{1,2}^{l-1} + w_{1,2}a_{1,3}^{l-1} + w_{2,1}a_{2,2}^{l-1} + w_{2,2}a_{2,3}^{l-1} + w_b \\ net_{2,1}^j &= w_{1,1}a_{2,1}^{l-1} + w_{1,2}a_{2,2}^{l-1} + w_{2,1}a_{3,1}^{l-1} + w_{2,2}a_{3,2}^{l-1} + w_b \\ net_{2,2}^j &= w_{1,1}a_{2,2}^{l-1} + w_{1,2}a_{2,3}^{l-1} + w_{2,1}a_{3,2}^{l-1} + w_{2,2}a_{3,3}^{l-1} + w_b \end{aligned}$$

从上面的公式看出，由于权值共享，权值  $w_{1,1}$  对所有的  $net_{i,j}^l$  都有影响。 $E_d$  是  $net_{1,1}^l$ 、 $net_{1,2}^l$ 、 $net_{2,1}^l$ ... 的函数，而  $net_{1,1}^l$ 、 $net_{1,2}^l$ 、 $net_{2,1}^l$ ... 又是  $w_{1,1}$  的函数，根据全导数公式，计算  $\frac{\partial E_d}{\partial w_{1,1}}$  就是要把每个偏导数都加起来：

$$\begin{aligned} \frac{\partial E_d}{\partial w_{1,1}} &= \frac{\partial E_d}{\partial net_{1,1}^l} \frac{\partial net_{1,1}^l}{\partial w_{1,1}} + \frac{\partial E_d}{\partial net_{1,2}^l} \frac{\partial net_{1,2}^l}{\partial w_{1,1}} + \frac{\partial E_d}{\partial net_{2,1}^l} \frac{\partial net_{2,1}^l}{\partial w_{1,1}} + \frac{\partial E_d}{\partial net_{2,2}^l} \frac{\partial net_{2,2}^l}{\partial w_{1,1}} \\ &= \delta_{1,1}^l a_{1,1}^{l-1} + \delta_{1,2}^l a_{1,2}^{l-1} + \delta_{2,1}^l a_{2,1}^{l-1} + \delta_{2,2}^l a_{2,2}^{l-1} \end{aligned}$$

**例 4.5** 计算  $\frac{\partial E_d}{\partial w_{1,2}}$ :

通过查看  $w_{1,2}$  与  $net_{i,j}^l$  的关系，我们很容易得到：

$$\frac{\partial E_d}{\partial w_{1,2}} = \delta_{1,1}^l a_{1,2}^{l-1} + \delta_{1,2}^l a_{1,3}^{l-1} + \delta_{2,1}^l a_{2,2}^{l-1} + \delta_{2,2}^l a_{2,3}^{l-1}$$

实际上，每个权重项都是类似的，我们不一一举例了。现在，是我们再次发挥想象力的时候，我们发现计算  $\frac{\partial E_d}{\partial w_{i,j}}$  规律是：

$$\frac{\partial E_d}{\partial w_{i,j}} = \sum_m \sum_n \delta_{m,n} a_{i+m, j+n}^{l-1}$$

也就是用 sensitivity map 作为卷积核，在 input 上进行 cross-correlation，如图4.17所示。

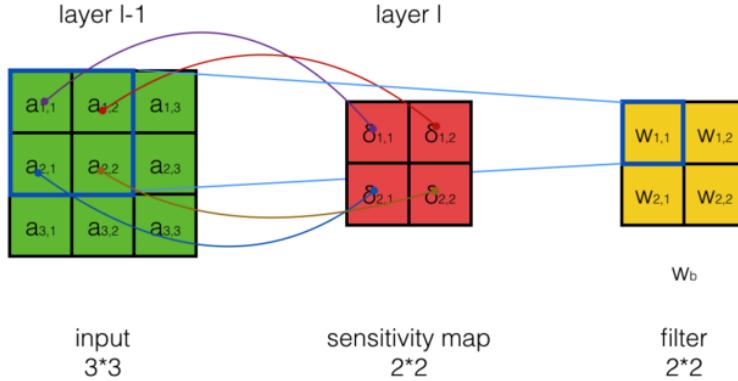


图 4.17: cross correlation

最后，我们来看一看偏置项的梯度  $\frac{\partial E_d}{\partial w_b}$ 。通过查看前面的公式，我们很容易发现：

$$\begin{aligned}\frac{\partial E_d}{\partial w_b} &= \frac{\partial E_d}{\partial net_{1,1}^l} \frac{\partial net_{1,1}^l}{\partial w_b} + \frac{\partial E_d}{\partial net_{1,2}^l} \frac{\partial net_{1,2}^l}{\partial w_b} + \frac{\partial E_d}{\partial net_{2,1}^l} \frac{\partial net_{2,1}^l}{\partial w_b} + \frac{\partial E_d}{\partial net_{2,2}^l} \frac{\partial net_{2,2}^l}{\partial w_b} \\ &= \delta_{1,1}^l + \delta_{1,2}^l + \delta_{2,1}^l + \delta_{2,2}^l = \sum_i \sum_j \delta_{i,j}^l\end{aligned}$$

也就是偏置项的梯度就是 sensitivity map 所有误差项之和。

对于步长为 S 的卷积层，处理方法与传递误差项是一样的，首先将 sensitivity map 『还原』成步长为 1 时的 sensitivity map，再用上面的方法进行计算。

获得了所有的梯度之后，就是根据梯度下降算法来更新每个权重。这在前面的文章中已经反复写过，这里就不再重复了。

至此，我们已经解决了卷积层的训练问题，接下来我们看一看 Pooling 层的训练。

### 4.5.2 Pooling 层的训练

无论 max pooling 还是 mean pooling，都没有需要学习的参数。因此，在卷积神经网络的训练中，Pooling 层需要做的仅仅是将误差项传递到上一层，而没有梯度的计算。

#### Max Pooling 误差项的传递

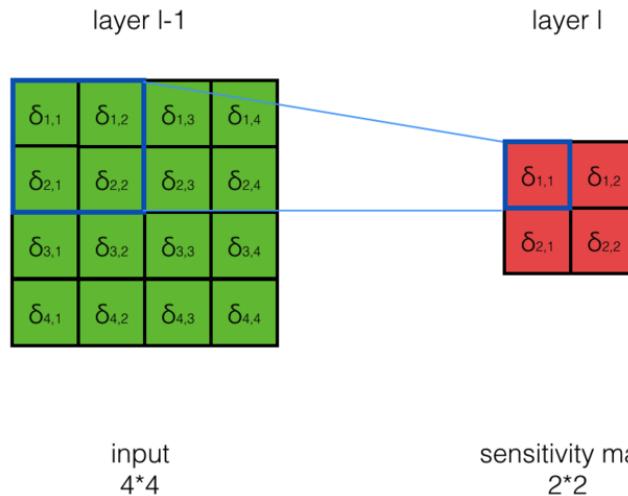


图 4.18: cross correlation

如图4.18，假设第 $l-1$ 层大小为 $4*4$ ，pooling filter 大小为 $2*2$ ，步长为2，这样，max pooling之后，第 $l$ 层大小为 $2*2$ 。假设第 $l$ 层的 $\delta$ 值都已经计算完毕，我们现在的任务是计算第 $l-1$ 层的 $\delta$ 值。

我们用 $net_{i,j}^{l-1}$ 表示第 $l-1$ 层的加权输入；用 $net_{i,j}^l$ 表示第 $l$ 层的加权输入。我们先来考察一个具体的例子，然后再总结一般性的规律。对于 max pooling：

$$net_{1,1}^l = \max(net_{1,1}^{l-1}, net_{1,2}^{l-1}, net_{2,1}^{l-1}, net_{2,2}^{l-1})$$

也就是说，只有区块中最大的 $net_{i,j}^{l-1}$ 才会对 $net_{i,j}^l$ 的值产生影响。我们假设最大的值是 $net_{1,1}^{l-1}$ ，则上式相当于：

$$net_{1,1}^l = net_{1,1}^{l-1}$$

那么，我们不难求得下面几个偏导数：

$$\begin{aligned}\frac{\partial net_{1,1}^l}{\partial net_{1,1}^{l-1}} &= 1, & \frac{\partial net_{1,1}^l}{\partial net_{1,2}^{l-1}} &= 0 \\ \frac{\partial net_{1,1}^l}{\partial net_{2,1}^{l-1}} &= 0, & \frac{\partial net_{1,1}^l}{\partial net_{2,2}^{l-1}} &= 0\end{aligned}$$

因此：

$$\begin{aligned}\delta_{1,1}^{l-1} &= \frac{\partial E_d}{\partial net_{1,1}^{l-1}} = \frac{\partial E_d}{\partial net_{1,1}^l} \frac{\partial net_{1,1}^l}{\partial net_{1,1}^{l-1}} = \delta_{1,1}^l, & \delta_{1,2}^{l-1} &= \frac{\partial E_d}{\partial net_{1,2}^{l-1}} = \frac{\partial E_d}{\partial net_{1,1}^l} \frac{\partial net_{1,1}^l}{\partial net_{1,2}^{l-1}} = 0 \\ \delta_{2,1}^{l-1} &= \frac{\partial E_d}{\partial net_{2,1}^{l-1}} = \frac{\partial E_d}{\partial net_{1,1}^l} \frac{\partial net_{1,1}^l}{\partial net_{2,1}^{l-1}} = 0, & \delta_{1,1}^{l-1} &= \frac{\partial E_d}{\partial net_{2,2}^{l-1}} = \frac{\partial E_d}{\partial net_{1,1}^l} \frac{\partial net_{1,1}^l}{\partial net_{2,2}^{l-1}} = 0\end{aligned}$$

现在，我们发现了规律：对于 max pooling，下一层的误差项的值会原封不动的传递到上一层对应区块中的最大值所对应的神经元，而其他神经元的误差项的值都是0。如图4.19所示（假设 $a_{1,1}^{l-1}$ 、 $a_{1,4}^{l-1}$ 、 $a_{4,1}^{l-1}$ 、 $a_{4,4}^{l-1}$ 为所在区块中的最大输出值）。

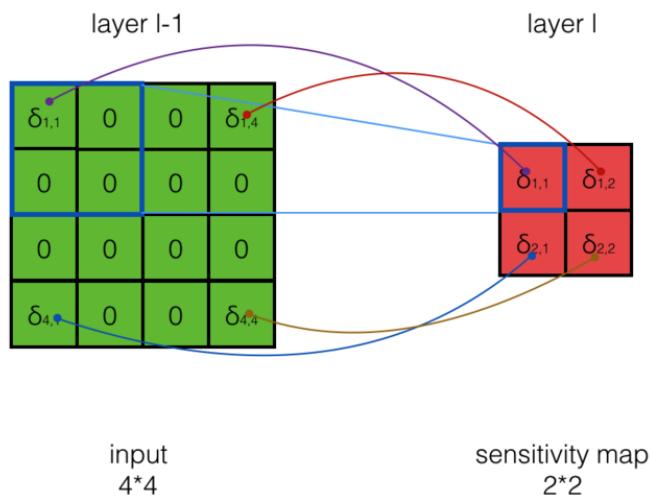


图 4.19: max pooling

### Mean Pooling 误差项的传递

我们还是用前面屡试不爽的套路，先研究一个特殊的情形，再扩展为一般规律。如

图4.20，我们先来考虑计算  $\delta_{1,1}^{l-1}$ 。我们先来看看  $net_{1,1}^{l-1}$  如何影响  $net_{1,1}^l$ 。

$$net_{1,1}^l = \frac{1}{4}(net_{1,1}^{l-1} + net_{1,2}^{l-1} + net_{2,1}^{l-1} + net_{2,2}^{l-1})$$

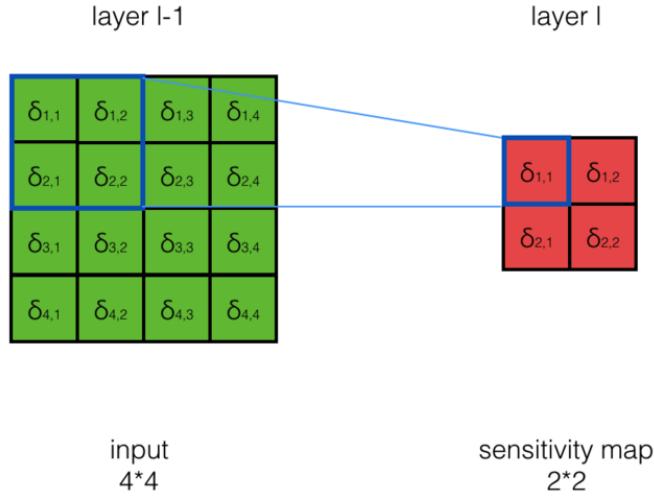


图 4.20: Mean Pooling

根据上式，我们一眼就能看出来：

$$\frac{\partial net_{1,1}^l}{\partial net_{1,1}^{l-1}} = \frac{1}{4} \frac{\partial net_{1,1}^l}{\partial net_{1,2}^{l-1}} = \frac{1}{4} \frac{\partial net_{1,1}^l}{\partial net_{2,1}^{l-1}} = \frac{1}{4} \frac{\partial net_{1,1}^l}{\partial net_{2,2}^{l-1}} = \frac{1}{4}$$

所以，根据链式求导法则，我们不难算出：

$$\begin{aligned}\delta_{1,1}^{l-1} &= \frac{\partial E_d}{\partial net_{1,1}^{l-1}} = \frac{\partial E_d}{\partial net_{1,1}^l} \frac{\partial net_{1,1}^l}{\partial net_{1,1}^{l-1}} = \frac{1}{4} \delta_{1,1}^l, & \delta_{1,2}^{l-1} &= \frac{\partial E_d}{\partial net_{1,2}^{l-1}} = \frac{\partial E_d}{\partial net_{1,1}^l} \frac{\partial net_{1,1}^l}{\partial net_{1,2}^{l-1}} = \frac{1}{4} \delta_{1,1}^l \\ \delta_{2,1}^{l-1} &= \frac{\partial E_d}{\partial net_{2,1}^{l-1}} = \frac{\partial E_d}{\partial net_{1,1}^l} \frac{\partial net_{1,1}^l}{\partial net_{2,1}^{l-1}} = \frac{1}{4} \delta_{1,1}^l, & \delta_{2,2}^{l-1} &= \frac{\partial E_d}{\partial net_{2,2}^{l-1}} = \frac{\partial E_d}{\partial net_{1,1}^l} \frac{\partial net_{1,1}^l}{\partial net_{2,2}^{l-1}} = \frac{1}{4} \delta_{1,1}^l\end{aligned}$$

现在，我们发现了规律：对于 mean pooling，下一层的误差项的值会平均分配到上一层对应区块中的所有神经元。如图4.21所示。

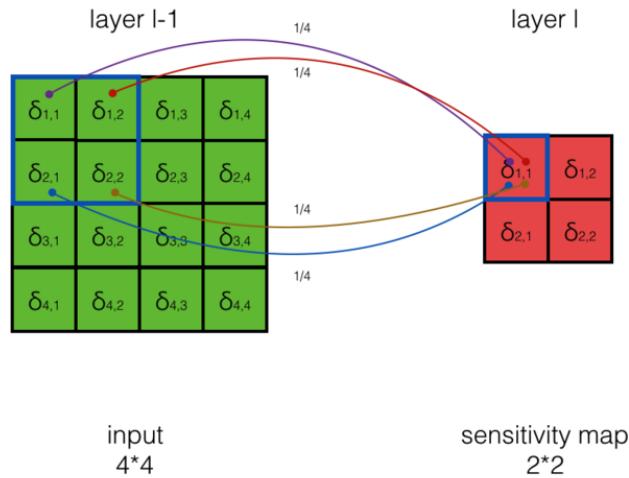


图 4.21: Mean Pooling

上面这个算法可以表达为高大上的克罗内克积 (Kronecker product) 的形式，有兴趣

的读者可以研究一下。

$$\delta^{l-1} = \delta^l \otimes \left(\frac{1}{n^2}\right)_{n \times n}$$

其中,  $n$  是 pooling 层 filter 的大小,  $\delta^{l-1}$ 、 $\delta^l$  都是矩阵。

至此, 我们已经把卷积层、Pooling 层的训练算法介绍完毕, 加上上一篇文章讲的全连接层训练算法, 您应该已经具备了编写卷积神经网络代码所需要的知识。为了加深对知识的理解, 接下来, 我们将展示如何实现一个简单的卷积神经网络。

## 4.6 编程实战：卷积神经网络的实现



**注意** 完整代码请参考 GitHub: [https://github.com/hanbt/learn\\_dl/blob/master/cnn.py](https://github.com/hanbt/learn_dl/blob/master/cnn.py) (python2.7)

现在, 我们亲自动手实现一个卷积神经网络, 以便巩固我们所学的知识。

首先, 我们要改变一下代码的架构, 『层』成为了我们最核心的组件。这是因为卷积神经网络有不同的层, 而每种层的算法都在对应的类中实现。

这次, 我们用到了在 python 中编写算法经常会用到的 **numpy** 包。为了使用 **numpy**, 我们需要先将 **numpy** 导入:

```
1 import numpy as np
```

### 4.6.1 卷积层的实现

#### 卷积层初始化

我们用 **ConvLayer** 类来实现一个卷积层。下面的代码是初始化一个卷积层, 可以在构造函数中设置卷积层的超参数。

```
1 class ConvLayer(object):
2     def __init__(self, input_width, input_height,
3                  channel_number, filter_width,
4                  filter_height, filter_number,
5                  zero_padding, stride, activator,
6                  learning_rate):
7         self.input_width = input_width
8         self.input_height = input_height
9         self.channel_number = channel_number
10        self.filter_width = filter_width
11        self.filter_height = filter_height
12        self.filter_number = filter_number
13        self.zero_padding = zero_padding
14        self.stride = stride
15        self.output_width =
16            ConvLayer.calculate_output_size(
17                self.input_width, filter_width, zero_padding,
```

```
18         stride)
19     self.output_height = \
20         ConvLayer.calculate_output_size(
21             self.input_height, filter_height, zero_padding,
22             stride)
23     self.output_array = np.zeros((self.filter_number,
24             self.output_height, self.output_width))
25     self.filters = []
26     for i in range(filter_number):
27         self.filters.append(Filter(filter_width,
28             filter_height, self.channel_number))
29     self.activator = activator
30     self.learning_rate = learning_rate
```

`calculate_output_size` 函数用来确定卷积层输出的大小，其实现如下：

```
1 @staticmethod
2 def calculate_output_size(input_size,
3     filter_size, zero_padding, stride):
4     return (input_size - filter_size +
5         2 * zero_padding) / stride + 1
```

**Filter** 类保存了卷积层的参数以及梯度，并且实现了用梯度下降算法来更新参数。

```
1 class Filter(object):
2     def __init__(self, width, height, depth):
3         self.weights = np.random.uniform(-1e-4, 1e-4,
4             (depth, height, width))
5         self.bias = 0
6         self.weights_grad = np.zeros(
7             self.weights.shape)
8         self.bias_grad = 0
9     def __repr__(self):
10        return 'filter weights:\n%s\nbias:\n%s' % (
11            repr(self.weights), repr(self.bias))
12     def get_weights(self):
13         return self.weights
14     def get_bias(self):
15         return self.bias
16     def update(self, learning_rate):
17         self.weights -= learning_rate * self.weights_grad
18         self.bias -= learning_rate * self.bias_grad
```

我们对参数的初始化采用了常用的策略，即：权重随机初始化为一个很小的值，而偏置项初始化为 0。

**Activator** 类实现了激活函数，其中，**forward** 方法实现了前向计算，而 **backward** 方法则是计算导数。比如，**relu** 函数的实现如下：

```
1 class ReluActivator(object):
2     def forward(self, weighted_input):
3         #return weighted_input
4         return max(0, weighted_input)
5     def backward(self, output):
6         return 1 if output > 0 else 0
```

### 卷积层前向计算的实现

**ConvLayer** 类的 **forward** 方法实现了卷积层的前向计算（即计算根据输入来计算卷积层的输出），下面是代码实现：

```
1     def forward(self, input_array):
2         ...
3
4         计算卷积层的输出
5         输出结果保存在 self.output_array
6         ...
7
8         self.input_array = input_array
9         self.padded_input_array = padding(input_array,
10             self.zero_padding)
11        for f in range(self.filter_number):
12            filter = self.filters[f]
13            conv(self.padded_input_array,
14                  filter.get_weights(), self.output_array[f],
15                  self.stride, filter.get_bias())
16            element_wise_op(self.output_array,
17                            self.activator.forward)
```

上面的代码里面包含了几个工具函数。**element\_wise\_op** 函数实现了对 **numpy** 数组进行按元素操作，并将返回值写回到数组中，代码如下：

```
1 # 对 numpy 数组进行 element wise 操作
2 def element_wise_op(array, op):
3     for i in np.nditer(array,
4                         op_flags=[ 'readwrite' ]):
5         i[...] = op(i)
```

**conv** 函数实现了 2 维和 3 维数组的卷积，代码如下：

```
1 def conv(input_array,
2          kernel_array,
3          output_array,
4          stride, bias):
5      ...
6
7      计算卷积，自动适配输入为 2D 和 3D 的情况
```

```
7     ''
8     channel_number = input_array.ndim
9     output_width = output_array.shape[1]
10    output_height = output_array.shape[0]
11    kernel_width = kernel_array.shape[-1]
12    kernel_height = kernel_array.shape[-2]
13    for i in range(output_height):
14        for j in range(output_width):
15            output_array[i][j] = (
16                get_patch(input_array, i, j, kernel_width,
17                           kernel_height, stride) * kernel_array
18            ).sum() + bias
```

**padding** 函数实现了 zero padding 操作:

```
1 # 为数组增加Zero padding
2 def padding(input_array, zp):
3     ''
4     为数组增加Zero padding，自动适配输入为2D和3D的情况
5     ''
6     if zp == 0:
7         return input_array
8     else:
9         if input_array.ndim == 3:
10             input_width = input_array.shape[2]
11             input_height = input_array.shape[1]
12             input_depth = input_array.shape[0]
13             padded_array = np.zeros((
14                 input_depth,
15                 input_height + 2 * zp,
16                 input_width + 2 * zp))
17             padded_array[:, zp : zp + input_height,
18                         zp : zp + input_width] = input_array
19             return padded_array
20         elif input_array.ndim == 2:
21             input_width = input_array.shape[1]
22             input_height = input_array.shape[0]
23             padded_array = np.zeros((
24                 input_height + 2 * zp,
25                 input_width + 2 * zp))
26             padded_array[zp : zp + input_height,
27                         zp : zp + input_width] = input_array
28             return padded_array
```

### 卷积层反向传播算法的实现

现在，是介绍卷积层核心算法的时候了。我们知道反向传播算法需要完成几个任务：

1. 将误差项传递到上一层。
2. 计算每个参数的梯度。
3. 更新参数。

以下代码都是在 **ConvLayer** 类中实现。我们先来看看将误差项传递到上一层的代码实现。

```
1 def bp_sensitivity_map(self, sensitivity_array,
2                         activator):
3     ...
4     计算传递到上一层的 sensitivity map
5     sensitivity_array: 本层的 sensitivity map
6     activator: 上一层的激活函数
7     ...
8     # 处理卷积步长，对原始 sensitivity map 进行扩展
9     expanded_array = self.expand_sensitivity_map(
10         sensitivity_array)
11    # full 卷积，对 sensitivity map 进行 zero padding
12    # 虽然原始输入的 zero padding 单元也会获得残差
13    # 但这个残差不需要继续向上传递，因此就不计算了
14    expanded_width = expanded_array.shape[2]
15    zp = (self.input_width +
16          self.filter_width - 1 - expanded_width) / 2
17    padded_array = padding(expanded_array, zp)
18    # 初始化 delta_array，用于保存传递到上一层的
19    # sensitivity map
20    self.delta_array = self.create_delta_array()
21    # 对于具有多个 filter 的卷积层来说，最终传递到上一层的
22    # sensitivity map 相当于所有的 filter 的
23    # sensitivity map 之和
24    for f in range(self.filter_number):
25        filter = self.filters[f]
26        # 将 filter 权重翻转 180 度
27        flipped_weights = np.array(map(
28            lambda i: np.rot90(i, 2),
29            filter.get_weights())))
30        # 计算与一个 filter 对应的 delta_array
31        delta_array = self.create_delta_array()
32        for d in range(delta_array.shape[0]):
33            conv(padded_array[f], flipped_weights[d],
34                  delta_array[d], 1, 0)
35        self.delta_array += delta_array
```

```
36     # 将计算结果与激活函数的偏导数做 element-wise 乘法操作
37     derivative_array = np.array(self.input_array)
38     element_wise_op(derivative_array,
39                      activator.backward)
40     self.delta_array *= derivative_array
```

**expand\_sensitivity\_map** 方法就是将步长为 S 的 sensitivity map『还原』为步长为 1 的 sensitivity map，代码如下：

```
1 def expand_sensitivity_map(self, sensitivity_array):
2     depth = sensitivity_array.shape[0]
3     # 确定扩展后 sensitivity map 的大小
4     # 计算 stride 为 1 时 sensitivity map 的大小
5     expanded_width = (self.input_width -
6                         self.filter_width + 2 * self.zero_padding + 1)
7     expanded_height = (self.input_height -
8                         self.filter_height + 2 * self.zero_padding + 1)
9     # 构建新的 sensitivity map
10    expand_array = np.zeros((depth, expanded_height,
11                             expanded_width))
12    # 从原始 sensitivity map 拷贝误差值
13    for i in range(self.output_height):
14        for j in range(self.output_width):
15            i_pos = i * self.stride
16            j_pos = j * self.stride
17            expand_array[:, i_pos, j_pos] = \
18                sensitivity_array[:, i, j]
19    return expand_array
```

**create\_delta\_array** 是创建用来保存传递到上一层的 sensitivity map 的数组。

```
1 def create_delta_array(self):
2     return np.zeros((self.channel_number,
3                      self.input_height, self.input_width))
```

接下来，是计算梯度的代码。

```
1 def bp_gradient(self, sensitivity_array):
2     # 处理卷积步长，对原始 sensitivity map 进行扩展
3     expanded_array = self.expand_sensitivity_map(
4         sensitivity_array)
5     for f in range(self.filter_number):
6         # 计算每个权重的梯度
7         filter = self.filters[f]
8         for d in range(filter.weights.shape[0]):
9             conv(self.padded_input_array[d],
10                  expanded_array[f],
```

```
1             filter.weights_grad[d], 1, 0)
2     # 计算偏置项的梯度
3     filter.bias_grad = expanded_array[f].sum()
```

最后，是按照梯度下降算法更新参数的代码，这部分非常简单。

```
1 def update(self):
2     ...
3     按照梯度下降，更新权重
4     ...
5     for filter in self.filters:
6         filter.update(self.learning_rate)
```

### 卷积层的梯度检查

为了验证我们的公式推导和代码实现的正确性，我们必须要对卷积层进行梯度检查。

下面是代码实现：

```
1 def init_test():
2     a = np.array(
3         [[[0, 1, 1, 0, 2],
4           [2, 2, 2, 2, 1],
5           [1, 0, 0, 2, 0],
6           [0, 1, 1, 0, 0],
7           [1, 2, 0, 0, 2]],
8           [[1, 0, 2, 2, 0],
9             [0, 0, 0, 2, 0],
10            [1, 2, 1, 2, 1],
11            [1, 0, 0, 0, 0],
12            [1, 2, 1, 1, 1]],
13            [[2, 1, 2, 0, 0],
14              [1, 0, 0, 1, 0],
15              [0, 2, 1, 0, 1],
16              [0, 1, 2, 2, 2],
17              [2, 1, 0, 0, 1]]])
18     b = np.array(
19         [[[0, 1, 1],
20           [2, 2, 2],
21           [1, 0, 0]],
22           [[1, 0, 2],
23             [0, 0, 0],
24             [1, 2, 1]]])
25     cl = ConvLayer(5, 5, 3, 3, 2, 1, 2, IdentityActivator(), 0.001)
26     cl.filters[0].weights = np.array(
27         [[[-1, 1, 0],
28           [0, 1, 0],
```



```
29         [0,1,1]] ,  
30         [[-1,-1,0],  
31         [0,0,0],  
32         [0,-1,0]],  
33         [[0,0,-1],  
34         [0,1,0],  
35         [1,-1,-1]]], dtype=np.float64)  
36     cl.filters[0].bias=1  
37     cl.filters[1].weights = np.array(  
38         [[[1,1,-1],  
39             [-1,-1,1],  
40             [0,-1,1]],  
41             [[0,1,0],  
42             [-1,0,-1],  
43             [-1,1,0]],  
44             [[-1,0,0],  
45             [-1,0,1],  
46             [-1,0,0]]], dtype=np.float64)  
47     return a, b, cl  
48 def gradient_check():  
49     ''',  
50     梯度检查  
51     ''',  
52     # 设计一个误差函数，取所有节点输出项之和  
53     error_function = lambda o: o.sum()  
54     # 计算forward值  
55     a, b, cl = init_test()  
56     cl.forward(a)  
57     # 求取sensitivity map，是一个全1数组  
58     sensitivity_array = np.ones(cl.output_array.shape,  
59                                     dtype=np.float64)  
60     # 计算梯度  
61     cl.backward(a, sensitivity_array,  
62                 IdentityActivator())  
63     # 检查梯度  
64     epsilon = 10e-4  
65     for d in range(cl.filters[0].weights_grad.shape[0]):  
66         for i in range(cl.filters[0].weights_grad.shape[1]):  
67             for j in range(cl.filters[0].weights_grad.shape[2]):  
68                 cl.filters[0].weights[d,i,j] += epsilon  
69                 cl.forward(a)  
70                 err1 = error_function(cl.output_array)  
71                 cl.filters[0].weights[d,i,j] -= 2*epsilon
```

```

72         cl.forward(a)
73         err2 = error_function(cl.output_array)
74         expect_grad = (err1 - err2) / (2 * epsilon)
75         cl.filters[0].weights[d,i,j] += epsilon
76         print 'weights(%d,%d,%d): expected - actural %f -'
77             %f' % (
d, i, j, expect_grad, cl.filters[0].
weights_grad[d,i,j])

```

上面代码值得思考的地方在于，传递给卷积层的 sensitivity map 是全 1 数组，留给读者自己推导一下为什么是这样（提示：激活函数选择了 identity 函数： $f(x) = x$ ）。读者如果还有困惑，请写在文章评论中，我会回复。

运行上面梯度检查的代码，我们得到的输出如下，期望的梯度和实际计算出的梯度一致，这证明我们的算法推导和代码实现确实是正确的。

```

>>> cnn.gradient_check()
weights(0,0,0): expected - actural 5.000000 - 5.000000
weights(0,0,1): expected - actural 6.000000 - 6.000000
weights(0,0,2): expected - actural 5.000000 - 5.000000
weights(0,1,0): expected - actural 5.000000 - 5.000000
weights(0,1,1): expected - actural 7.000000 - 7.000000
weights(0,1,2): expected - actural 5.000000 - 5.000000
weights(0,2,0): expected - actural 5.000000 - 5.000000
weights(0,2,1): expected - actural 6.000000 - 6.000000
weights(0,2,2): expected - actural 5.000000 - 5.000000
weights(1,0,0): expected - actural 2.000000 - 2.000000
weights(1,0,1): expected - actural 1.000000 - 1.000000
weights(1,0,2): expected - actural 2.000000 - 2.000000
weights(1,1,0): expected - actural 9.000000 - 9.000000
weights(1,1,1): expected - actural 9.000000 - 9.000000
weights(1,1,2): expected - actural 9.000000 - 9.000000
weights(1,2,0): expected - actural 2.000000 - 2.000000
weights(1,2,1): expected - actural 1.000000 - 1.000000
weights(1,2,2): expected - actural 2.000000 - 2.000000
weights(2,0,0): expected - actural 4.000000 - 4.000000
weights(2,0,1): expected - actural 5.000000 - 5.000000
weights(2,0,2): expected - actural 4.000000 - 4.000000

```

以上就是卷积层的实现。

## 4.6.2 Max Pooling 层的实现

max pooling 层的实现相对简单，我们直接贴出全部代码如下：

```

1 class MaxPoolingLayer(object):
2     def __init__(self, input_width, input_height,
3                  channel_number, filter_width,
4                  filter_height, stride):
5         self.input_width = input_width
6         self.input_height = input_height
7         self.channel_number = channel_number
8         self.filter_width = filter_width

```

```
9         self.filter_height = filter_height
10        self.stride = stride
11        self.output_width = (input_width -
12            filter_width) / self.stride + 1
13        self.output_height = (input_height -
14            filter_height) / self.stride + 1
15        self.output_array = np.zeros((self.channel_number,
16            self.output_height, self.output_width))
17    def forward(self, input_array):
18        for d in range(self.channel_number):
19            for i in range(self.output_height):
20                for j in range(self.output_width):
21                    self.output_array[d,i,j] = (
22                        get_patch(input_array[d], i, j,
23                            self.filter_width,
24                            self.filter_height,
25                            self.stride).max())
26    def backward(self, input_array, sensitivity_array):
27        self.delta_array = np.zeros(input_array.shape)
28        for d in range(self.channel_number):
29            for i in range(self.output_height):
30                for j in range(self.output_width):
31                    patch_array = get_patch(
32                        input_array[d], i, j,
33                        self.filter_width,
34                        self.filter_height,
35                        self.stride)
36                    k, l = get_max_index(patch_array)
37                    self.delta_array[d,
38                        i * self.stride + k,
39                        j * self.stride + l] = \
40                        sensitivity_array[d,i,j]
```

全连接层的实现和上一篇文章类似，在此就不再赘述了。至此，你已经拥有了实现了一个简单的卷积神经网络所需要的基本组件。对于卷积神经网络，现在有很多优秀的开源实现，因此我们并不需要真的自己去实现一个。贴出这些代码的目的是为了让我们更好的了解卷积神经网络的基本原理。

## 4.7 卷积神经网络的应用

### MNIST 手写数字识别

*LeNet-5* 是实现手写数字识别的卷积神经网络，在 MNIST 测试集上，它取得了 0.8% 的错误率。*LeNet-5* 的结构如图4.22：



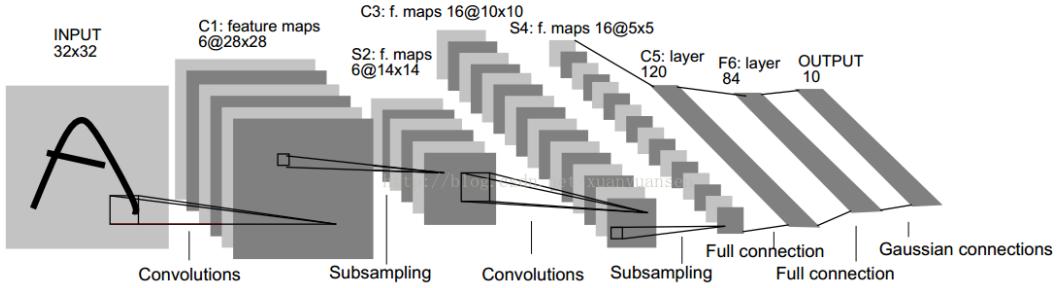


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

图 4.22: LeNet-5

关于 *LeNet-5* 的详细介绍，网上的资料很多，因此就不再重复了。感兴趣的读者可以尝试用我们自己实现的卷积神经网络代码去构造并训练 *LeNet-5*（当然代码会更复杂一些）。

## 4.8 小结

由于卷积神经网络的复杂性，我们写出了整个系列目前为止最长的一篇文章，相信读者也和作者一样累的要死。卷积神经网络是深度学习最重要的工具（我犹豫要不要写上『之一』呢），付出一些辛苦去理解它也是值得的。如果您真正理解了本文的内容，相当于迈过了入门深度学习最重要的一道门槛。在下一篇文章中，我们介绍深度学习另外一种非常重要的工具：循环神经网络，届时我们的系列文章也将完成过半。每篇文章都是一个过滤器，对于坚持到这里的读者们，入门深度学习曙光已现，加油。

# 第 5 章 循环神经网络

## 内容提要

- 语言模型 5.1
- 循环神经网络是啥 5.2
- 基本循环神经网络 5.2.1
- 双向循环神经网络 5.2.2
- 深度循环神经网络 5.2.3
- 循环神经网络的训练 5.3
- 训练算法：BPTT 5.3.1
- 梯度爆炸和消失问题 5.3.2
- RNN 的应用：语言模型 5.4
- 向量化 5.4.1
- Softmax 层 5.4.2
- 语言模型的训练 5.4.3
- 交叉熵误差 5.4.4
- 编程实战：RNN 的实现 5.5

在前面的文章系列文章中，我们介绍了全连接神经网络和卷积神经网络，以及它们的训练和使用。他们都只能单独的取处理一个个的输入，前一个输入和后一个输入是完全没有关系的。但是，某些任务需要能够更好的处理序列的信息，即前面的输入和后面的输入是有关系的。比如，当我们在理解一句话意思时，孤立的理解这句话的每个词是不够的，我们需要处理这些词连接起来的整个序列；当我们处理视频的时候，我们也不能只单独的去分析每一帧，而要分析这些帧连接起来的整个序列。这时，就需要用到深度学习领域中另一类非常重要神经网络：**循环神经网络 (Recurrent Neural Network)**。RNN 种类很多，也比较绕脑子。不过读者不用担心，本文将一如既往的对复杂的东西剥茧抽丝，帮助您理解 RNNs 以及它的训练算法，并动手实现一个循环神经网络。

## 5.1 语言模型

RNN 是在自然语言处理领域中最先被用起来的，比如，RNN 可以为语言模型来建模。那么，什么是语言模型呢？

我们可以和电脑玩一个游戏，我们写出一个句子前面的一些词，然后，让电脑帮我们写下接下来的一个词。比如下面这句：

我昨天上学迟到了，老师批评了 - - - - -。

我们给电脑展示了这句话前面这些词，然后，让电脑写下接下来的一个词。在这个例子中，接下来的这个词最有可能是『我』，而不太可能是『小明』，甚至是『吃饭』。

语言模型就是这样的东西：给定一个一句话前面的部分，预测接下来最有可能的一个词是什么。

语言模型是对一种语言的特征进行建模，它有很多很多用处。比如在语音转文本 (STT) 的应用中，声学模型输出的结果，往往是若干个可能的候选词，这时候就需要语言模型来从这些候选词中选择一个最可能的。当然，它同样也可以用在图像到文本的识别中 (OCR)。

使用 RNN 之前，语言模型主要是采用 N-Gram。N 可以是一个自然数，比如 2 或者 3。它的含义是，假设一个词出现的概率只与前面 N 个词相关。我们以 2-Gram 为例。首先，对前面的一句话进行切词：

我 昨 天 上 学 迟 到 了 ， 老 师 批 评 了 - - - - -。

如果用 2-Gram 进行建模，那么电脑在预测的时候，只会看到前面的『了』，然后，电脑会在语料库中，搜索『了』后面最可能的一个词。不管最后电脑选的是不是『我』，我们都应该知道这个模型是不靠谱的，因为『了』前面说了那么一大堆实际上是没有用到的。如果是 3-Gram 模型呢，会搜索『批评了』后面最可能的词，感觉上比 2-Gram 靠谱了不少，但还是远远不够的。因为这句话最关键的信息『我』，远在 9 个词之前！

现在读者可能会想，可以提升继续提升 N 的值呀，比如 4-Gram、5-Gram.....。实际上，这个想法是没有实用性的。因为我们想处理任意长度的句子，N 设为多少都不合适；另外，模型的大小和 N 的关系是指数级的，4-Gram 模型就会占用海量的存储空间。

所以，该轮到 RNN 出场了，RNN 理论上可以往前看(往后看)任意多个词。

## 5.2 循环神经网络是啥

循环神经网络种类繁多，我们先从最简单的基本循环神经网络开始吧。

### 5.2.1 基本循环神经网络

图5.1是一个简单的循环神经网络如，它由输入层、一个隐藏层和一个输出层组成：

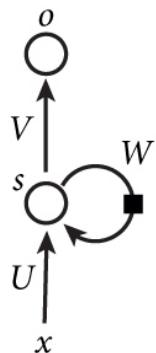


图 5.1：简单的循环神经网络

纳尼？！相信第一次看到这个玩意的读者内心和我一样是崩溃的。因为循环神经网络实在是太难画出来了，网上所有大神们都不得不用了这种抽象艺术手法。不过，静下心来仔细看看的话，其实也是很好理解的。如果把上面有 W 的那个带箭头的圈去掉，它就变成了最普通的全连接神经网络。 $x$  是一个向量，它表示输入层的值（这里面没有画出来表示神经元节点的圆圈）； $s$  是一个向量，它表示隐藏层的值（这里隐藏层面画了一个节点，你也可以想象这一层其实是多个节点，节点数与向量  $s$  的维度相同）； $U$  是输入层到隐藏层的权重矩阵（读者可以回到第3章神经网络和反向传播算法，看看我们是怎样用矩阵来表示全连接神经网络的计算的）； $o$  也是一个向量，它表示输出层的值； $V$  是隐藏

层到输出层的权重矩阵。那么，现在我们来看看  $W$  是什么。循环神经网络的隐藏层的值  $s$  不仅仅取决于当前这次的输入  $x$ ，还取决于上一次隐藏层的值  $s$ 。权重矩阵  $W$  就是隐藏层上一次的值作为这一次的输入的权重。

如果我们把上面的图展开，循环神经网络也可以画成图5.2这个样子：

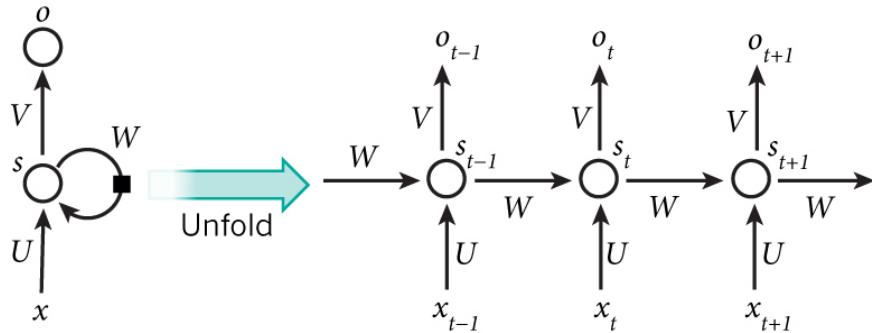


图 5.2: 循环神经网络

现在看上去就比较清楚了，这个网络在  $t$  时刻接收到输入  $x_t$  之后，隐藏层的值是  $s_t$ ，输出值是  $o_t$ 。关键一点是， $s_t$  的值不仅仅取决于  $x_t$ ，还取决于  $s_{t-1}$ 。我们可以用下面的公式来表示循环神经网络的计算方法：

$$o_t = g(Vs_t) \quad (5.1)$$

$$s_t = f(Ux_t + Ws_{t-1}) \quad (5.2)$$

公式 5.1 是输出层的计算公式，输出层是一个全连接层，也就是它的每个节点都和隐藏层的每个节点相连。 $V$  是输出层的权重矩阵， $g$  是激活函数。公式 5.2 是隐藏层的计算公式，它是循环层。 $U$  是输入  $x$  的权重矩阵， $W$  是上一次的值  $s_{t-1}$  作为这一次的输入的权重矩阵， $f$  是激活函数。

从上面的公式我们可以看出，循环层和全连接层的区别就是循环层多了一个权重矩阵  $W$ 。

如果反复把公式5.2带入到公式5.1，我们将得到：

$$\begin{aligned} o_t &= g(Vs_t) = Vf(Ux_t + Ws_{t-1}) = g\left(Vf\left(Ux_t + Wf(Ux_{t-1} + Ws_{t-2})\right)\right) \\ &= g\left(Vf\left(Ux_t + Wf\left(Ux_{t-1} + Wf(Ux_{t-2} + Ws_{t-3})\right)\right)\right) \\ &= g\left(Vf\left(Ux_t + Wf\left(Ux_{t-1} + Wf\left(Ux_{t-2} + Wf(Ux_{t-3} + \dots)\right)\right)\right)\right) \end{aligned}$$

从上面可以看出，循环神经网络的输出值  $o_t$ ，是受前面历次输入值  $x_t$ 、 $x_{t-1}$ 、 $x_{t-2}$ 、 $x_{t-3}$ 、... 影响的，这就是为什么循环神经网络可以往前看任意多个输入值的原因。

## 5.2.2 双向循环神经网络

对于语言模型来说，很多时候光看前面的词是不够的，比如下面这句话：

我的手机坏了，我打算 - - - 一部新手机。

可以想象，如果我们只看横线前面的词，手机坏了，那么我是打算修一修？换一部新的？还是大哭一场？这些都是无法确定的。但如果我们也看到了横线后面的词是『一部新手机』，那么，横线上的词填『买』的概率就大得多了。

在上一小节中的基本循环神经网络是无法对此进行建模的，因此，我们需要双向循环神经网络，如图5.3所示。

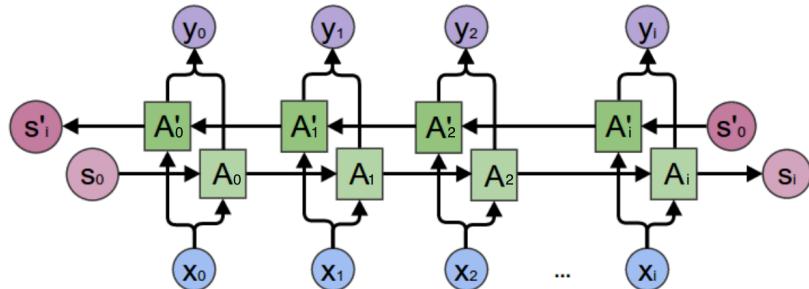


图 5.3：双向循环神经网络

当遇到这种从未来穿越回来的场景时，难免处于懵逼的状态。不过我们还是可以用屡试不爽的老办法：先分析一个特殊场景，然后再总结一般规律。我们先考虑图5.3中， $y_2$ 的计算。

从图5.3可以看出，双向卷积神经网络的隐藏层要保存两个值，一个  $A$  参与正向计算，另一个值  $A'$  参与反向计算。最终的输出值  $y_2$  取决于  $A_2$  和  $A'_2$ 。其计算方法为：

$$y_2 = g(VA_2 + V'A'_2)$$

$A_2$  和  $A'_2$  则分别计算：

$$A_2 = f(WA_1 + UX_2)$$

$$A'_2 = f(W'A'_3 + U'X_2)$$

现在，我们已经可以看出一般的规律：正向计算时，隐藏层的值  $s_t$  与  $s_{t-1}$  有关；反向计算时，隐藏层的值  $s'_t$  与  $s'_{t+1}$  有关；最终的输出取决于正向和反向计算的加和。现在，我们仿照公式5.1和5.2，写出双向循环神经网络的计算方法：

$$o_t = g(Vs_t + V's'_t)$$

$$s_t = f(Ux_t + Ws_{t-1})$$

$$s'_t = f(U'x_t + W's'_{t+1})$$

从上面三个公式我们可以看到，正向计算和反向计算不共享权重，也就是说  $U$  和  $U'$ 、 $W$  和  $W'$ 、 $V$  和  $V'$  都是不同的权重矩阵。

### 5.2.3 深度循环神经网络

前面我们介绍的循环神经网络只有一个隐藏层，我们当然也可以堆叠两个以上的隐藏层，这样就得到了深度循环神经网络。如图5.4所示。

我们把第  $i$  个隐藏层的值表示为  $s_t^{(i)}$ 、 $s'_t^{(i)}$ ，则深度循环神经网络的计算方式可以表

示为：

$$\begin{aligned}
 o_t &= g(V^{(i)} s_t^{(i)} + V'^{(i)} s_t'^{(i)}) \\
 s_t^{(i)} &= f(U^{(i)} s_t^{(i-1)} + W^{(i)} s_{t-1}) \\
 s_t'^{(i)} &= f(U'^{(i)} s_t'^{(i-1)} + W'^{(i)} s_{t+1}') \\
 &\dots \\
 s_t^{(1)} &= f(U^{(1)} x_t + W^{(1)} s_{t-1}) \\
 s_t'^{(1)} &= f(U'^{(1)} x_t + W'^{(1)} s_{t+1})
 \end{aligned}$$

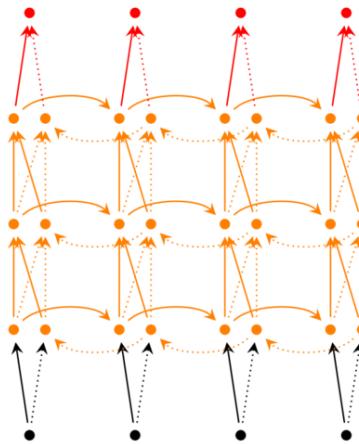


图 5.4: 深度循环神经网络

## 5.3 循环神经网络的训练

### 5.3.1 循环神经网络的训练算法：BPTT

BPTT 算法是针对循环层的训练算法，它的基本原理和 BP 算法是一样的，也包含同样的三个步骤：

1. 前向计算每个神经元的输出值；
2. 反向计算每个神经元的误差项  $\delta_j$  值，它是误差函数  $E$  对神经元  $j$  的加权输入  $net_j$  的偏导数；
3. 计算每个权重的梯度。

最后再用随机梯度下降算法更新权重。

循环层如图5.5所示：

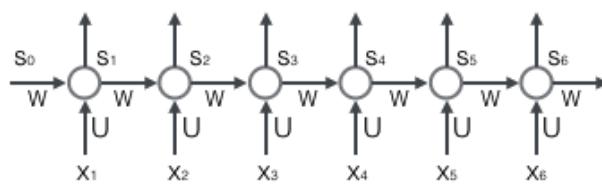


图 5.5: 循环层

## 前向计算

使用前面的公式5.2对循环层进行前向计算：

$$s_t = f(Ux_t + Ws_{t-1})$$

注意，上面的  $s_t$ 、 $x_t$ 、 $s_{t-1}$  都是向量，用黑体字母表示；而  $U$ 、 $V$  是矩阵，用大写字母表示。向量的下标表示时刻，例如， $s_t$  表示在  $t$  时刻向量  $s$  的值。

我们假设输入向量  $x$  的维度是  $m$ ，输出向量  $s$  的维度是  $n$ ，则矩阵  $U$  的维度是  $n \times m$ ，矩阵  $W$  的维度是  $n \times n$ 。下面是上式展开成矩阵的样子，看起来更直观一些：

$$\begin{bmatrix} s_1^t \\ s_2^t \\ \vdots \\ s_n^t \end{bmatrix} = f \left( \begin{bmatrix} u_{11}u_{12}\dots u_{1m} \\ u_{21}u_{22}\dots u_{2m} \\ \vdots \\ u_{n1}u_{n2}\dots u_{nm} \end{bmatrix} \begin{bmatrix} x_1^t \\ x_2^t \\ \vdots \\ x_m^t \end{bmatrix} + \begin{bmatrix} w_{11}w_{12}\dots w_{1n} \\ w_{21}w_{22}\dots w_{2n} \\ \vdots \\ w_{n1}w_{n2}\dots w_{nn} \end{bmatrix} \begin{bmatrix} s_1^{t-1} \\ s_2^{t-1} \\ \vdots \\ s_n^{t-1} \end{bmatrix} \right)$$

在这里我们用手写体字母表示向量的一个元素，它的下标表示它是这个向量的第几个元素，它的上标表示第几个时刻。例如， $s_j^t$  表示向量  $s$  的第  $j$  个元素在  $t$  时刻的值。 $u_{ji}$  表示输入层第  $i$  个神经元到循环层第  $j$  个神经元的权重。 $w_{ji}$  表示循环层第  $t-1$  时刻的第  $i$  个神经元到循环层第  $t$  个时刻的第  $j$  个神经元的权重。

## 误差项的计算

BTTP 算法将第  $l$  层  $t$  时刻的误差项  $\delta_t^l$  值沿两个方向传播，一个方向是其传递到上一层网络，得到  $\delta_t^{l-1}$ ，这部分只和权重矩阵  $U$  有关；另一个方向是将其沿时间线传递到初始  $t_1$  时刻，得到  $\delta_{t_1}^l$ ，这部分只和权重矩阵  $W$  有关。

我们用向量  $net_t$  表示神经元在  $t$  时刻的加权输入，因为：

$$net_t = Ux_t + Ws_{t-1}$$

$$s_{t-1} = f(net_{t-1})$$

因此：

$$\frac{\partial net_t}{\partial net_{t-1}} = \frac{\partial net_t}{\partial s_{t-1}} \frac{\partial s_{t-1}}{\partial net_{t-1}}$$

我们用  $a$  表示列向量，用  $a^T$  表示行向量。上式的第一项是向量函数对向量求导，其结果为 Jacobian 矩阵：

$$\frac{\partial net_t}{\partial s_{t-1}} = \begin{bmatrix} \frac{\partial net_1^t}{\partial s_1^{t-1}} & \frac{\partial net_1^t}{\partial s_2^{t-1}} & \dots & \frac{\partial net_1^t}{\partial s_n^{t-1}} \\ \frac{\partial net_2^t}{\partial s_1^{t-1}} & \frac{\partial net_2^t}{\partial s_2^{t-1}} & \dots & \frac{\partial net_2^t}{\partial s_n^{t-1}} \\ \vdots & & & \\ \frac{\partial net_n^t}{\partial s_1^{t-1}} & \frac{\partial net_n^t}{\partial s_2^{t-1}} & \dots & \frac{\partial net_n^t}{\partial s_n^{t-1}} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & & & \\ w_{n1} & w_{n2} & \dots & w_{nn} \end{bmatrix} = W$$

同理，上式第二项也是一个 Jacobian 矩阵：

$$\begin{aligned}\frac{\partial s_{t-1}}{\partial net_{t-1}} &= \begin{bmatrix} \frac{\partial s_1^{t-1}}{\partial net_1^{t-1}} & \frac{\partial s_1^{t-1}}{\partial net_2^{t-1}} & \cdots & \frac{\partial s_1^{t-1}}{\partial net_n^{t-1}} \\ \frac{\partial s_2^{t-1}}{\partial net_1^{t-1}} & \frac{\partial s_2^{t-1}}{\partial net_2^{t-1}} & \cdots & \frac{\partial s_2^{t-1}}{\partial net_n^{t-1}} \\ \vdots & & & \\ \frac{\partial s_n^{t-1}}{\partial net_1^{t-1}} & \frac{\partial s_n^{t-1}}{\partial net_2^{t-1}} & \cdots & \frac{\partial s_n^{t-1}}{\partial net_n^{t-1}} \end{bmatrix} \\ &= \begin{bmatrix} f'(net_1^{t-1}) & 0 & \cdots & 0 \\ 0 & f'(net_2^{t-1}) & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & f'(net_n^{t-1}) \end{bmatrix} \\ &= diag[f'(net_{t-1})]\end{aligned}$$

其中， $diag(a)$  表示根据向量  $a$  创建一个对角矩阵，即

$$diag(a) = \begin{bmatrix} a_1 & 0 & \cdots & 0 \\ 0 & a_2 & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & a_n \end{bmatrix}$$

最后，将两项合在一起，可得：

$$\begin{aligned}\frac{\partial net_t}{\partial net_{t-1}} &= \frac{\partial net_t}{\partial s_{t-1}} \frac{\partial s_{t-1}}{\partial net_{t-1}} = W diag[f'(net_{t-1})] \\ &= \begin{bmatrix} w_{11}f'(net_1^{t-1}) & w_{12}f'(net_2^{t-1}) & \cdots & w_{1n}f'(net_n^{t-1}) \\ w_{21}f'(net_1^{t-1}) & w_{22}f'(net_2^{t-1}) & \cdots & w_{2n}f'(net_n^{t-1}) \\ \vdots & & & \\ w_{n1}f'(net_1^{t-1}) & w_{n2}f'(net_2^{t-1}) & \cdots & w_{nn}f'(net_n^{t-1}) \end{bmatrix}\end{aligned}$$

上式描述了将  $\delta$  沿时间往前传递一个时刻的规律，有了这个规律，我们就可以求得任意时刻  $k$  的误差项  $\delta_k$ ：

$$\begin{aligned}\delta_k^T &= \frac{\partial E}{\partial net_k} = \frac{\partial E}{\partial net_t} \frac{\partial net_t}{\partial net_k} = \frac{\partial E}{\partial net_t} \frac{\partial net_t}{\partial net_{t-1}} \frac{\partial net_{t-1}}{\partial net_{t-2}} \cdots \frac{\partial net_{k+1}}{\partial net_k} \\ &= W diag[f'(net_{t-1})] W diag[f'(net_{t-2})] \cdots W diag[f'(net_k)] \delta_t^l \\ &= \delta_t^T \prod_{i=k}^{t-1} W diag[f'(net_i)]\end{aligned}\tag{5.3}$$

公式5.3就是将误差项沿时间反向传播的算法。

循环层将误差项反向传递到上一层网络，与普通的全连接层是完全一样的，这在第3章神经网络和反向传播算法中已经详细讲过了，在此仅简要描述一下。

循环层的加权输入  $net^l$  与上一层的加权输入  $net^{l-1}$  关系如下：

$$\begin{aligned}net_t^l &= U a_t^{l-1} + W s_{t-1} \\ a_t^{l-1} &= f^{l-1}(net_t^{l-1})\end{aligned}$$

上式中  $net_t^l$  是第  $l$  层神经元的加权输入（假设第 1 层是循环层）； $net_t^{l-1}$  是第  $l-1$  层神经

元的加权输入； $a_t^{l-1}$  是第  $l-1$  层神经元的输出； $f^{l-1}$  是第  $l-1$  层的激活函数。

$$\frac{\partial \text{net}_t^l}{\partial \text{net}_t^{l-1}} = \frac{\partial \text{net}^l}{\partial a_t^{l-1}} \frac{\partial a_t^{l-1}}{\partial \text{net}_t^{l-1}} = U \text{diag}[f'^{l-1}(\text{net}_t^{l-1})]$$

所以

$$\begin{aligned} (\delta_t^{l-1})^T &= \frac{\partial E}{\partial \text{net}_t^{l-1}} = \frac{\partial E}{\partial \text{net}_t^l} \frac{\partial \text{net}_t^l}{\partial \text{net}_t^{l-1}} \\ &= (\delta_t^l)^T U \text{diag}[f'^{l-1}(\text{net}_t^{l-1})] \end{aligned} \quad (5.4)$$

公式5.4就是将误差项传递到上一层算法。

### 权重梯度的计算

现在，我们终于来到了 BPTT 算法的最后一步：计算每个权重的梯度。

首先，我们计算误差函数  $E$  对权重矩阵  $W$  的梯度  $\frac{\partial E}{\partial W}$ 。

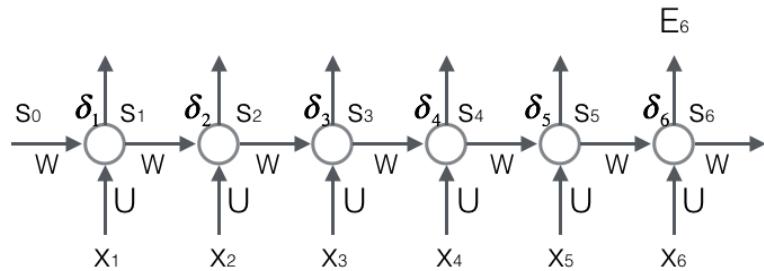


图 5.6: 权重梯度

图5.6展示了我们到目前为止，在前两步中已经计算得到的量，包括每个时刻  $t$  循环层的输出值  $s_t$ ，以及误差项  $\delta_t$ 。

回忆一下我们在第3章神经网络和反向传播算法介绍的全连接网络的权重梯度计算算法：只要知道了任意一个时刻的误差项  $\delta_t$ ，以及上一个时刻循环层的输出值  $s_{t-1}$ ，就可以按照下面的公式求出权重矩阵在  $t$  时刻的梯度  $\nabla_{W_t} E$ ：

$$\nabla_{W_t} E = \begin{bmatrix} \delta_1^t s_1^{t-1} & \delta_1^t s_2^{t-1} & \dots & \delta_1^t s_n^{t-1} \\ \delta_2^t s_1^{t-1} & \delta_2^t s_2^{t-1} & \dots & \delta_2^t s_n^{t-1} \\ \vdots & & & \\ \delta_n^t s_1^{t-1} & \delta_n^t s_2^{t-1} & \dots & \delta_n^t s_n^{t-1} \end{bmatrix} \quad (5.5)$$

在公式5.5中， $\delta_i^t$  表示  $t$  时刻误差项向量的第  $i$  个分量； $s_i^{t-1}$  表示  $t-1$  时刻循环层第  $i$  个神经元的输出值。

我们下面可以简单推导一下公式5.5。

我们知道：

$$\begin{aligned}
 net_t &= Ux_t + Ws_{t-1} \\
 \begin{bmatrix} net_1^t \\ net_2^t \\ \vdots \\ net_n^t \end{bmatrix} &= Ux_t + \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & & & \\ w_{n1} & w_{n2} & \dots & w_{nn} \end{bmatrix} \begin{bmatrix} s_1^{t-1} \\ s_2^{t-1} \\ \vdots \\ s_n^{t-1} \end{bmatrix} \\
 &= Ux_t + \begin{bmatrix} w_{11}s_1^{t-1} + w_{12}s_2^{t-1} + \dots + w_{1n}s_n^{t-1} \\ w_{21}s_1^{t-1} + w_{22}s_2^{t-1} + \dots + w_{2n}s_n^{t-1} \\ \vdots \\ w_{n1}s_1^{t-1} + w_{n2}s_2^{t-1} + \dots + w_{nn}s_n^{t-1} \end{bmatrix}
 \end{aligned}$$

因为对  $W$  求导与  $Ux_t$  无关，我们不再考虑。现在，我们考虑对权重项  $w_{ji}$  求导。通过观察上式我们可以看到  $w_{ji}$  只与  $net_j^t$  有关，所以：

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial net_j^t} \frac{\partial net_j^t}{\partial w_{ji}} = \delta_j^t s_i^{t-1}$$

按照上面的规律就可以生成公式5.5里面的矩阵。

我们已经求得了权重矩阵  $W$  在  $t$  时刻的梯度  $\nabla_{Wt}E$ ，最终的梯度  $\nabla_W E$  是各个时刻的梯度之和：

$$\begin{aligned}
 \nabla_W E &= \sum_{i=1}^t \nabla_{W_i} E \\
 &= \begin{bmatrix} \delta_1^t s_1^{t-1} & \delta_1^t s_2^{t-1} & \dots & \delta_1^t s_n^{t-1} \\ \delta_2^t s_1^{t-1} & \delta_2^t s_2^{t-1} & \dots & \delta_2^t s_n^{t-1} \\ \vdots & & & \\ \delta_n^t s_1^{t-1} & \delta_n^t s_2^{t-1} & \dots & \delta_n^t s_n^{t-1} \end{bmatrix} + \dots + \begin{bmatrix} \delta_1^1 s_1^0 & \delta_1^1 s_2^0 & \dots & \delta_1^1 s_n^0 \\ \delta_2^1 s_1^0 & \delta_2^1 s_2^0 & \dots & \delta_2^1 s_n^0 \\ \vdots & & & \\ \delta_n^1 s_1^0 & \delta_n^1 s_2^0 & \dots & \delta_n^1 s_n^0 \end{bmatrix} \quad (5.6)
 \end{aligned}$$

公式5.6就是计算循环层权重矩阵  $W$  的梯度的公式。

### ——数学公式超高能预警——

前面已经介绍了  $\nabla_W E$  的计算方法，看上去还是比较直观的。然而，读者也许会困惑，为什么最终的梯度是各个时刻的梯度之和呢？我们前面只是直接用了这个结论，实际上这里面是有道理的，只是这个数学推导比较绕脑子。感兴趣的同學可以仔细阅读接下来这一段，它用到了矩阵对矩阵求导、张量与向量相乘运算的一些法则。

我们还是从这个式子开始：

$$net_t = Ux_t + Wf(net_{t-1})$$

因为  $Ux_t$  与  $W$  完全无关，我们把它看做常量。现在，考虑第一个式子加号右边的部分，因为  $W$  和  $f(net_{t-1})$  都是  $W$  的函数，因此我们要用到大学里面都学过的导数乘法运算：

$$(uv)' = u'v + uv'$$

因此，上面第一个式子写成：

$$\frac{\partial \text{net}_t}{\partial W} = \frac{\partial W}{\partial W} f(\text{net}_{t-1}) + W \frac{\partial f(\text{net}_{t-1})}{\partial W}$$

我们最终需要计算的是  $\nabla_W E$ ：

$$\begin{aligned} \nabla_W E &= \frac{\partial E}{\partial W} = \frac{\partial E}{\partial \text{net}_t} \frac{\partial \text{net}_t}{\partial W} \\ &= \delta_t^T \frac{\partial W}{\partial W} f(\text{net}_{t-1}) + \delta_t^T W \frac{\partial f(\text{net}_{t-1})}{\partial W} \end{aligned} \quad (5.7)$$

我们先计算公式5.7加号左边的部分。 $\frac{\partial W}{\partial W}$  是矩阵对矩阵求导，其结果是一个四维张量 (tensor)，如下所示：

$$\begin{aligned} \frac{\partial W}{\partial W} &= \begin{bmatrix} \frac{\partial w_{11}}{\partial W} & \frac{\partial w_{12}}{\partial W} & \cdots & \frac{\partial w_{1n}}{\partial W} \\ \frac{\partial w_{21}}{\partial W} & \frac{\partial w_{22}}{\partial W} & \cdots & \frac{\partial w_{2n}}{\partial W} \\ \vdots & & & \\ \frac{\partial w_{n1}}{\partial W} & \frac{\partial w_{n2}}{\partial W} & \cdots & \frac{\partial w_{nn}}{\partial W} \end{bmatrix} \\ &= \left[ \begin{bmatrix} \frac{\partial w_{11}}{\partial w_{11}} & \frac{\partial w_{11}}{\partial w_{12}} & \cdots & \frac{\partial w_{11}}{\partial w_{1n}} \\ \frac{\partial w_{11}}{\partial w_{21}} & \frac{\partial w_{11}}{\partial w_{22}} & \cdots & \frac{\partial w_{11}}{\partial w_{2n}} \\ \vdots & & & \\ \frac{\partial w_{11}}{\partial w_{n1}} & \frac{\partial w_{11}}{\partial w_{n2}} & \cdots & \frac{\partial w_{11}}{\partial w_{nn}} \end{bmatrix} \begin{bmatrix} \frac{\partial w_{12}}{\partial w_{11}} & \frac{\partial w_{12}}{\partial w_{12}} & \cdots & \frac{\partial w_{12}}{\partial w_{1n}} \\ \frac{\partial w_{12}}{\partial w_{21}} & \frac{\partial w_{12}}{\partial w_{22}} & \cdots & \frac{\partial w_{12}}{\partial w_{2n}} \\ \vdots & & & \\ \frac{\partial w_{12}}{\partial w_{n1}} & \frac{\partial w_{12}}{\partial w_{n2}} & \cdots & \frac{\partial w_{12}}{\partial w_{nn}} \end{bmatrix} \cdots \right] \\ &= \left[ \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & 0 \\ \vdots & & & \end{bmatrix} \begin{bmatrix} 0 & 1 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & 0 \\ \vdots & & & \end{bmatrix} \cdots \right] \end{aligned}$$

接下来，我们知道  $s_{t-1} = f(\text{net}_{t-1})$ ，它是一个列向量。我们让上面的四维张量与这个向量相乘，得到了一个三维张量，再左乘行向量  $\delta_t^T$ ，最终得到一个矩阵：

$$\begin{aligned}
& \delta_t^T \frac{\partial W}{\partial W} f(\text{net}_{t-1}) = \delta_t^T \frac{\partial W}{\partial W} s_{t-1} = \delta_t^T \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & & & \\ 0 & 0 & \dots & 0 \\ \vdots & & & \end{bmatrix} \begin{bmatrix} 0 & 1 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & & & \\ 0 & 0 & \dots & 0 \\ \vdots & & & \end{bmatrix} \dots \begin{bmatrix} s_1^{t-1} \\ s_2^{t-1} \\ \vdots \\ s_n^{t-1} \end{bmatrix} \\
& = \delta_t^T \begin{bmatrix} s_1^{t-1} \\ 0 \\ \vdots \\ 0 \\ \vdots \end{bmatrix} \begin{bmatrix} s_2^{t-1} \\ 0 \\ \vdots \\ 0 \\ \vdots \end{bmatrix} \dots = \begin{bmatrix} \delta_1^t & \delta_2^t & \dots & \delta_n^t \end{bmatrix} \begin{bmatrix} s_1^{t-1} \\ 0 \\ \vdots \\ 0 \\ \vdots \end{bmatrix} \begin{bmatrix} s_2^{t-1} \\ 0 \\ \vdots \\ 0 \\ \vdots \end{bmatrix} \dots \\
& = \begin{bmatrix} \delta_1^t s_1^{t-1} & \delta_1^t s_2^{t-1} & \dots & \delta_1^t s_n^{t-1} \\ \delta_2^t s_1^{t-1} & \delta_2^t s_2^{t-1} & \dots & \delta_2^t s_n^{t-1} \\ \vdots & & & \\ \delta_n^t s_1^{t-1} & \delta_n^t s_2^{t-1} & \dots & \delta_n^t s_n^{t-1} \end{bmatrix} = \nabla_{Wt} E
\end{aligned}$$

接下来，我们计算公式5.7加号右边的部分：

$$\begin{aligned}
\delta_t^T W \frac{\partial f(\text{net}_{t-1})}{\partial W} &= \delta_t^T W \frac{\partial f(\text{net}_{t-1})}{\partial \text{net}_{t-1}} \frac{\partial \text{net}_{t-1}}{\partial W} = \delta_t^T W f'(\text{net}_{t-1}) \frac{\partial \text{net}_{t-1}}{\partial W} \\
&= \delta_t^T \frac{\partial \text{net}_t}{\partial \text{net}_{t-1}} \frac{\partial \text{net}_{t-1}}{\partial W} = \delta_{t-1}^T \frac{\partial \text{net}_{t-1}}{\partial W}
\end{aligned}$$

于是，我们得到了如下递推公式：

$$\begin{aligned}
\nabla_W E &= \frac{\partial E}{\partial W} = \frac{\partial E}{\partial \text{net}_t} \frac{\partial \text{net}_t}{\partial W} = \nabla_{Wt} E + \delta_{t-1}^T \frac{\partial \text{net}_{t-1}}{\partial W} \\
&= \nabla_{Wt} E + \nabla_{Wt-1} E + \delta_{t-2}^T \frac{\partial \text{net}_{t-2}}{\partial W} \\
&= \nabla_{Wt} E + \nabla_{Wt-1} E + \dots + \nabla_{W1} E \\
&= \sum_{k=1}^t \nabla_{Wk} E
\end{aligned}$$

这样，我们就证明了：最终的梯度  $\nabla_W E$  是各个时刻的梯度之和。

---数学公式超高能预警解除---

同权重矩阵  $W$  类似，我们可以得到权重矩阵  $U$  的计算方法。

$$\nabla_{U_t} E = \begin{bmatrix} \delta_1^t x_1^t & \delta_1^t x_2^t & \dots & \delta_1^t x_m^t \\ \delta_2^t x_1^t & \delta_2^t x_2^t & \dots & \delta_2^t x_m^t \\ \vdots & & & \\ \delta_n^t x_1^t & \delta_n^t x_2^t & \dots & \delta_n^t x_m^t \end{bmatrix} \quad (5.8)$$

公式5.8是误差函数在  $t$  时刻对权重矩阵  $U$  的梯度。和权重矩阵  $W$  一样，最终的梯

度也是各个时刻的梯度之和：

$$\nabla_U E = \sum_{i=1}^t \nabla_{U_i} E$$

具体的证明这里就不再赘述了，感兴趣的读者可以练习推导一下。

### 5.3.2 RNN 的梯度爆炸和消失问题

不幸的是，实践中前面介绍的几种 RNNs 并不能很好的处理较长的序列。一个主要的原因是，RNN 在训练中很容易发生**梯度爆炸**和**梯度消失**，这导致训练时梯度不能在较长序列中一直传递下去，从而使 RNN 无法捕捉到长距离的影响。

为什么 RNN 会产生梯度爆炸和消失问题呢？我们接下来将详细分析一下原因。我们根据公式5.3可得：

$$\begin{aligned}\delta_k^T &= \delta_t^T \prod_{i=k}^{t-1} W \text{diag}[f'(\text{net}_i)] \\ \|\delta_k^T\| &\leq \|\delta_t^T\| \prod_{i=k}^{t-1} \|W\| \|\text{diag}[f'(\text{net}_i)]\| \\ &\leq \|\delta_t^T\| (\beta_W \beta_f)^{t-k}\end{aligned}$$

上式的  $\beta$  定义为矩阵的模的上界。因为上式是一个指数函数，如果  $t - k$  很大的话（也就是向前看很远的时候），会导致对应的误差项的值增长或缩小的非常快，这样就会导致相应的**梯度爆炸**和**梯度消失**问题（取决于  $\beta$  大于 1 还是小于 1）。

通常来说，梯度爆炸更容易处理一些。因为梯度爆炸的时候，我们的程序会收到 *Nan* 错误。我们也可以设置一个梯度阈值，当梯度超过这个阈值的时候可以直接截取。

梯度消失更难检测，而且也更难处理一些。总的来说，我们有三种方法应对梯度消失问题：

1. 合理的初始化权重值。初始化权重，使每个神经元尽可能不要取极大或极小值，以躲开梯度消失的区域。
2. 使用 `relu` 代替 `sigmoid` 和 `tanh` 作为激活函数。原理请参考第4章卷积神经网络的**4.1 激活函数**一节。
3. 使用其他结构的 RNNs，比如长短时记忆网络(LSTM)和 Gated Recurrent Unit(GRU)，这是最流行的做法。我们将在以后的文章中介绍这两种网络。

## 5.4 RNN 的应用：基于 RNN 的语言模型

现在，我们介绍一下基于 RNN 语言模型。我们首先把词依次输入到循环神经网络中，每输入一个词，循环神经网络就输出截止到目前为止，下一个最可能的词。例如，当我们依次输入：

我 昨 天 上 学 迟 到 了

神经网络的输出如图5.7所示：



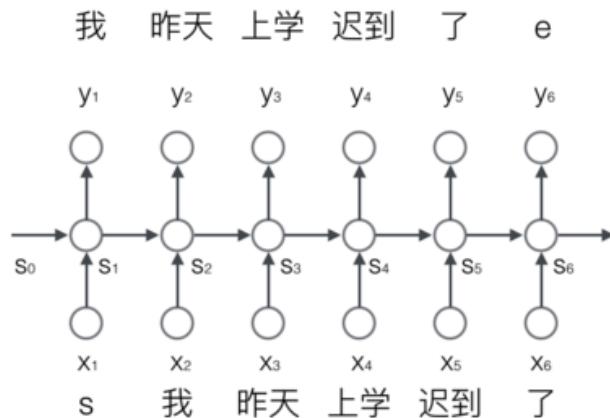


图 5.7: 神经网络的输出

其中， $s$  和  $e$  是两个特殊的词，分别表示一个序列的开始和结束。

#### 5.4.1 向量化

我们知道，神经网络的输入和输出都是向量，为了让语言模型能够被神经网络处理，我们必须把词表达为向量的形式，这样神经网络才能处理它。

神经网络的输入是词，我们可以用下面的步骤对输入进行向量化：

- 建立一个包含所有词的词典，每个词在词典里面有一个唯一的编号。
- 任意一个词都可以用一个  $N$  维的 one-hot 向量来表示。其中， $N$  是词典中包含的词的个数。假设一个词在词典中的编号是  $i$ ， $v$  是表示这个词的向量， $v_j$  是向量的第  $j$  个元素，则：

$$v_j = \begin{cases} 1 & j = i \\ 0 & otherwise \end{cases}$$

上面这个公式的含义，可以用图5.8来直观的表示。使用这种向量化方法，我们就得

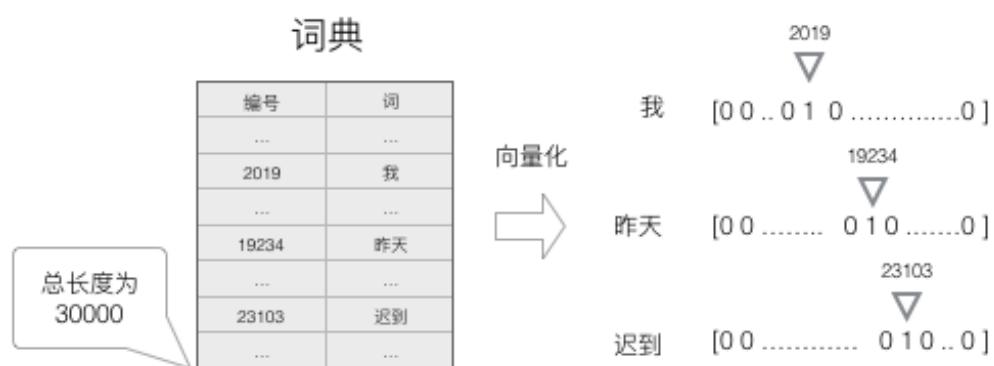


图 5.8: 向量化

到了一个高维、稀疏的向量（稀疏是指绝大部分元素的值都是 0）。处理这样的向量会导致我们的神经网络有很多的参数，带来庞大的计算量。因此，往往需要使用一些降维方法，将高维的稀疏向量转变为低维的稠密向量。不过这个话题我们就不再这篇文章中讨论了。

语言模型要求的输出是下一个最可能的词，我们可以让循环神经网络计算计算词典中每个词是下一个词的概率，这样，概率最大的词就是下一个最可能的词。因此，神经网络的输出向量也是一个  $N$  维向量，向量中的每个元素对应着词典中相应的词是下一个词的概率。如图5.9所示：



图 5.9：向量化

#### 5.4.2 Softmax 层

前面提到，语言模型是对下一个词出现的概率进行建模。那么，怎样让神经网络输出概率呢？方法就是用 softmax 层作为神经网络的输出层。

我们先来看一下 softmax 函数的定义：

$$g(z_i) = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

这个公式看起来可能很晕，我们举一个例子。Softmax 层如图5.10所示，从图5.10我们可以看到，softmax layer 的输入是一个向量，输出也是一个向量，两个向量的维度是一样的（在这个例子里面是 4）。输入向量  $x = [1, 2, 3, 4]$  经过 softmax 层之后，经过上面的 softmax 函数计算，转变为输出向量  $y = [0.03, 0.09, 0.24, 0.64]$ 。计算过程为：

$$\begin{aligned} y_1 &= \frac{e^{x_1}}{\sum_k e^{x_k}} = \frac{e^1}{e^1 + e^2 + e^3 + e^4} = 0.03 \\ y_2 &= \frac{e^2}{e^1 + e^2 + e^3 + e^4} = 0.09 \\ y_3 &= \frac{e^3}{e^1 + e^2 + e^3 + e^4} = 0.24 \\ y_4 &= \frac{e^4}{e^1 + e^2 + e^3 + e^4} = 0.64 \end{aligned}$$

我们来看看输出向量  $y$  的特征：

1. 每一项为取值为 0-1 之间的正数；
2. 所有项的总和是 1。

我们不难发现，这些特征和概率的特征是一样的，因此我们可以把它们看做是概率。对于语言模型来说，我们可以认为模型预测下一个词是词典中第一个词的概率是 0.03，是词典中第二个词的概率是 0.09，以此类推。

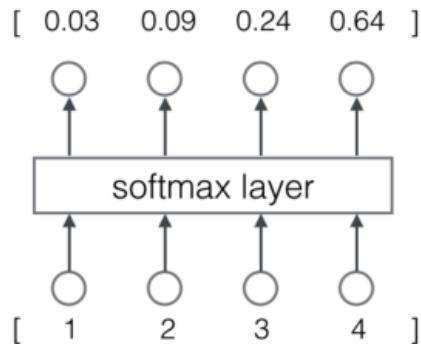


图 5.10: Softmax 层

### 5.4.3 语言模型的训练

可以使用监督学习的方法对语言模型进行训练，首先，需要准备训练数据集。接下来，我们介绍怎样把语料

我 昨 天 上 学 迟 到 了

转换成语言模型的训练数据集。

首先，我们获取输入-标签对：

表 5.1: 输入-标签

输入	标签
s	我
我	昨天
昨天	上学
上学	迟到
迟到	了
了	e

然后，使用前面介绍过的向量化方法，对输入  $x$  和标签  $y$  进行向量化。这里面有意思的是，对标签  $y$  进行向量化，其结果也是一个 one-hot 向量。例如，我们对标签『我』进行向量化，得到的向量中，只有第 2019 个元素的值是 1，其他位置的元素的值都是 0。它的含义就是下一个词是『我』的概率是 1，是其它词的概率都是 0。

最后，我们使用交叉熵误差函数作为优化目标，对模型进行优化。

在实际工程中，我们可以使用大量的语料来对模型进行训练，获取训练数据和训练的方法都是相同的。

### 5.4.4 交叉熵误差

一般来说，当神经网络的输出层是 softmax 层时，对应的误差函数  $S$  通常选择交叉熵误差函数，其定义如下：

$$L(y, o) = -\frac{1}{N} \sum_{n \in N} y_n \log o_n$$

在上式中， $N$  是训练样本的个数，向量  $y_n$  是样本的标记，向量  $o_n$  是网络的输出。标

记  $y_n$  是一个 one-hot 向量, 例如  $y_1 = [1, 0, 0, 0]$ , 如果网络的输出  $o = [0.03, 0.09, 0.24, 0.64]$ , 那么, 交叉熵误差是 (假设只有一个训练样本, 即  $N=1$ ):

$$\begin{aligned} L &= -\frac{1}{N} \sum_{n \in N} y_n \log o_n = -y_1 \log o_1 \\ &= -(1 * \log 0.03 + 0 * \log 0.09 + 0 * \log 0.24 + 0 * \log 0.64) = 3.51 \end{aligned}$$

我们当然可以选择其他函数作为我们的误差函数, 比如最小平方误差函数 (MSE)。不过对概率进行建模时, 选择交叉熵误差函数更 make sense。具体原因, 感兴趣的读者请阅读 (<https://jamesmccaffrey.wordpress.com/2011/12/17/neural-network-classification-categorical-data-softmax-activation-and-cross-entropy-error/>)。

## 5.5 编程实战: RNN 的实现

 **注意** 完整代码请参考 GitHub: [https://github.com/hanbt/learn\\_dl/blob/master/rnn.py](https://github.com/hanbt/learn_dl/blob/master/rnn.py) (python2.7)

为了加深我们对前面介绍的知识的理解, 我们来动手实现一个 RNN 层。我们复用了第4章卷积神经网络中的一些代码, 所以先把它们导入进来。

```
1 import numpy as np
2 from cnn import ReluActivator, IdentityActivator, element_wise_op
```

我们用 RecurrentLayer 类来实现一个循环层。下面的代码是初始化一个循环层, 可以在构造函数中设置卷积层的超参数。我们注意到, 循环层有两个权重数组,  $U$  和  $W$ 。

```
1 class RecurrentLayer(object):
2     def __init__(self, input_width, state_width,
3                  activator, learning_rate):
4         self.input_width = input_width
5         self.state_width = state_width
6         self.activator = activator
7         self.learning_rate = learning_rate
8         self.times = 0          # 当前时刻初始化为 t0
9         self.state_list = []    # 保存各个时刻的 state
10        self.state_list.append(np.zeros(
11            (state_width, 1)))      # 初始化 s0
12        self.U = np.random.uniform(-1e-4, 1e-4,
13            (state_width, input_width)) # 初始化 U
14        self.W = np.random.uniform(-1e-4, 1e-4,
15            (state_width, state_width)) # 初始化 W
```

在 forward 方法中, 实现循环层的前向计算, 这部分比较简单。

```
1     def forward(self, input_array):
2         ...
```

```
3     根据『式2』进行前向计算
4     ...
5     self.times += 1
6     state = (np.dot(self.U, input_array) +
7               np.dot(self.W, self.state_list[-1]))
8     element_wise_op(state, self.activator.forward)
9     self.state_list.append(state)
```

在 backward 方法中，实现 BPTT 算法。

```
1 def backward(self, sensitivity_array,
2               activator):
3     ...
4     实现BPTT算法
5     ...
6     self.calc_delta(sensitivity_array, activator)
7     self.calc_gradient()
8     def calc_delta(self, sensitivity_array, activator):
9         self.delta_list = [] # 用来保存各个时刻的误差项
10        for i in range(self.times):
11            self.delta_list.append(np.zeros(
12                (self.state_width, 1)))
13        self.delta_list.append(sensitivity_array)
14        # 迭代计算每个时刻的误差项
15        for k in range(self.times - 1, 0, -1):
16            self.calc_delta_k(k, activator)
17    def calc_delta_k(self, k, activator):
18        ...
19        根据k+1时刻的delta计算k时刻的delta
20        ...
21        state = self.state_list[k+1].copy()
22        element_wise_op(self.state_list[k+1],
23                         activator.backward)
24        self.delta_list[k] = np.dot(
25            np.dot(self.delta_list[k+1].T, self.W),
26            np.diag(state[:,0])).T
27    def calc_gradient(self):
28        self.gradient_list = [] # 保存各个时刻的权重梯度
29        for t in range(self.times + 1):
30            self.gradient_list.append(np.zeros(
31                (self.state_width, self.state_width)))
32        for t in range(self.times, 0, -1):
33            self.calc_gradient_t(t)
34        # 实际的梯度是各个时刻梯度之和
```

```

35         self.gradient = reduce(
36             lambda a, b: a + b, self.gradient_list,
37             self.gradient_list[0]) # [0]被初始化为0且没有被修改过
38     def calc_gradient_t(self, t):
39         ...
40         计算每个时刻t权重的梯度
41         ...
42         gradient = np.dot(self.delta_list[t],
43             self.state_list[t-1].T)
44         self.gradient_list[t] = gradient

```

有意思的是，BPTT 算法虽然数学推导的过程很麻烦，但是写成代码却并不复杂。在 update 方法中，实现梯度下降算法。

```

1 def update(self):
2     ...
3     按照梯度下降，更新权重
4     ...
5     self.W -= self.learning_rate * self.gradient

```

上面的代码不包含权重 U 的更新。这部分实际上和全连接神经网络是一样的，留给感兴趣的读者自己来完成吧。

循环层是一个带状态的层，每次 forward 都会改变循环层的内部状态，这给梯度检查带来了麻烦。因此，我们需要一个 reset\_state 方法，来重置循环层的内部状态。

```

1 def reset_state(self):
2     self.times = 0          # 当前时刻初始化为t0
3     self.state_list = []    # 保存各个时刻的state
4     self.state_list.append(np.zeros(
5         (self.state_width, 1)))    # 初始化s0

```

最后，是梯度检查的代码。

```

1 def gradient_check():
2     ...
3     梯度检查
4     ...
5     # 设计一个误差函数，取所有节点输出项之和
6     error_function = lambda o: o.sum()
7     rl = RecurrentLayer(3, 2, IdentityActivator(), 1e-3)
8     # 计算forward值
9     x, d = data_set()
10    rl.forward(x[0])
11    rl.forward(x[1])
12    # 求取sensitivity map
13    sensitivity_array = np.ones(rl.state_list[-1].shape,

```

```
14                                         dtype=np.float64)
15     # 计算梯度
16     rl.backward(sensitivity_array, IdentityActivator())
17     # 检查梯度
18     epsilon = 10e-4
19     for i in range(rl.W.shape[0]):
20         for j in range(rl.W.shape[1]):
21             rl.W[i,j] += epsilon
22             rl.reset_state()
23             rl.forward(x[0])
24             rl.forward(x[1])
25             err1 = error_function(rl.state_list[-1])
26             rl.W[i,j] -= 2*epsilon
27             rl.reset_state()
28             rl.forward(x[0])
29             rl.forward(x[1])
30             err2 = error_function(rl.state_list[-1])
31             expect_grad = (err1 - err2) / (2 * epsilon)
32             rl.W[i,j] += epsilon
33             print 'weights(%d,%d): expected - actural %f - %f' % (
34                 i, j, expect_grad, rl.gradient[i,j])
```

需要注意，每次计算 error 之前，都要调用 reset\_state 方法重置循环层的内部状态。

下面是梯度检查的结果，没问题！

```
>>> rnn.gradient_check()
weights(0,0): expected - actural 0.000013 - 0.000013
weights(0,1): expected - actural 0.000383 - 0.000383
weights(1,0): expected - actural 0.000013 - 0.000013
weights(1,1): expected - actural 0.000383 - 0.000383
```

## 5.6 小节

至此，我们讲完了基本的循环神经网络、它的训练算法：**BPTT**，以及在语言模型上的应用。RNN 比较烧脑，相信拿下前几篇文章的读者们搞定这篇文章也不在话下吧！然而，循环神经网络这个话题并没有完结。我们在前面说到过，基本的循环神经网络存在梯度爆炸和梯度消失问题，并不能真正的处理好长距离的依赖（虽然有一些技巧可以减轻这些问题）。事实上，真正得到广泛的应用的是循环神经网络的一个变体：**长短时记忆网络**。它内部有一些特殊的结构，可以很好的处理长距离的依赖，我们将在下一篇文章中详细的介绍它。现在，让我们稍事休息，准备挑战更为烧脑的**长短时记忆网络**吧。

# 第 6 章 长短时记忆网络

## 内容提要

- 长短时记忆网络是啥 6.1
- 长短时记忆网络的前向计算 6.2
- 长短时记忆网络的训练 6.3
- LSTM 训练算法框架 6.3.1
- 关于公式和符号的说明 6.3.2
- 误差项沿时间的反向传递 6.3.3
- 将误差项传递到上一层 6.3.4
- 权重梯度的计算 6.3.5
- 编程实战：长短时记忆网络的实
- 现 6.4
- 激活函数的实现 6.4.1
- LSTM 初始化 6.4.2
- 前向计算的实现 6.4.3
- 反向传播算法的实现 6.4.4
- 梯度下降算法的实现 6.4.5
- 梯度检查的实现 6.4.6
- GRU 6.5

在上一篇文章中，我们介绍了循环神经网络以及它的训练算法。我们也介绍了循环神经网络很难训练的原因，这导致了它在实际应用中，很难处理长距离的依赖。在本文中，我们将介绍一种改进之后的循环神经网络：**长短时记忆网络 (Long Short Term Memory Network, LSTM)**，它成功的解决了原始循环神经网络的缺陷，成为当前最流行的 RNN，在语音识别、图片描述、自然语言处理等许多领域中成功应用。但不幸的一面是，**LSTM** 的结构很复杂，因此，我们需要花上一些力气，才能把 LSTM 以及它的训练算法弄明白。在搞清楚 **LSTM** 之后，我们再介绍一种 **LSTM** 的变体：**GRU (Gated Recurrent Unit)**。它的结构比 **LSTM** 简单，而效果却和 **LSTM** 一样好，因此，它正在逐渐流行起来。最后，我们仍然会动手实现一个 **LSTM**。

## 6.1 长短时记忆网络是啥

我们首先了解一下长短时记忆网络产生的背景。回顾一下第4章循环神经网络中推导的，误差项沿时间反向传播的公式：

$$\delta_k^T = \delta_t^T \prod_{i=k}^{t-1} \text{diag}[f'(\text{net}_i)]W$$

我们可以根据下面的不等式，来获取  $\delta_k^T$  的模的上界（模可以看做对  $\delta_k^T$  中每一项值的大小的度量）：

$$\begin{aligned} \|\delta_k^T\| &\leq \|\delta_t^T\| \prod_{i=k}^{t-1} \|\text{diag}[f'(\text{net}_i)]\| \|W\| \\ &\leq \|\delta_t^T\| (\beta_f \beta_w)^{t-k} \end{aligned}$$

我们可以看到，误差项  $\delta$  从 t 时刻传递到 k 时刻，其值的上界是  $\beta_f \beta_w$  的指数函数。 $\beta_f \beta_w$  分别是对角矩阵  $\text{diag}[f'(\text{net}_i)]$  和矩阵 W 模的上界。显然，除非  $\beta_f \beta_w$  乘积的值位

于 1 附近，否则，当  $t-k$  很大时（也就是误差传递很多个时刻时），整个式子的值就会变得极小（当  $\beta_f \beta_w$  乘积小于 1）或者极大（当  $\beta_f \beta_w$  乘积大于 1），前者就是梯度消失，后者就是梯度爆炸。虽然科学家们搞出了很多技巧（比如怎样初始化权重），让  $\beta_f \beta_w$  的值尽可能贴近于 1，终究还是难以抵挡指数函数的威力。

梯度消失到底意味着什么？在第4章循环神经网络中我们已证明，权重数组  $W$  最终的梯度是各个时刻的梯度之和，即：

$$\nabla_W E = \sum_{k=1}^t \nabla_{W_k} E = \nabla_{W_t} E + \nabla_{W_{t-1}} E + \nabla_{W_{t-2}} E + \dots + \nabla_{W_1} E$$

假设某轮训练中，各时刻的梯度以及最终的梯度之和如图6.1：我们就可以看到，从

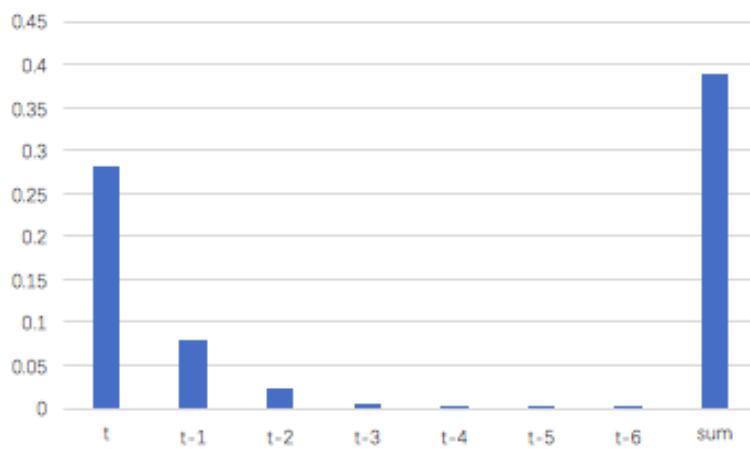


图 6.1: 梯度

上图的  $t-3$  时刻开始，梯度已经几乎减少到 0 了。那么，从这个时刻开始再往之前走，得到的梯度（几乎为零）就不会对最终的梯度值有任何贡献，这就相当于无论  $t-3$  时刻之前的网络状态  $h$  是什么，在训练中都不会对权重数组  $W$  的更新产生影响，也就是网络事实上已经忽略了  $t-3$  时刻之前的状态。这就是原始 RNN 无法处理长距离依赖的原因。

既然找到了问题的原因，那么我们就能解决它。从问题的定位到解决，科学家们大概花了 7、8 年时间。终于有一天，Hochreiter 和 Schmidhuber 两位科学家发明出长短时记忆网络，一举解决这个问题。

其实，长短时记忆网络的思路比较简单。原始 RNN 的隐藏层只有一个状态，即  $h$ ，它对于短期的输入非常敏感。那么，假如我们再增加一个状态，即  $c$ ，让它来保存长期的状态，那么问题不就解决了么？如图6.2所示：

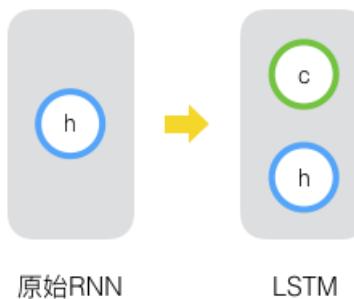


图 6.2: RNN to LSTM

新增加的状态  $c$ ，称为单元状态 (cell state)。我们把上图按照时间维度展开：

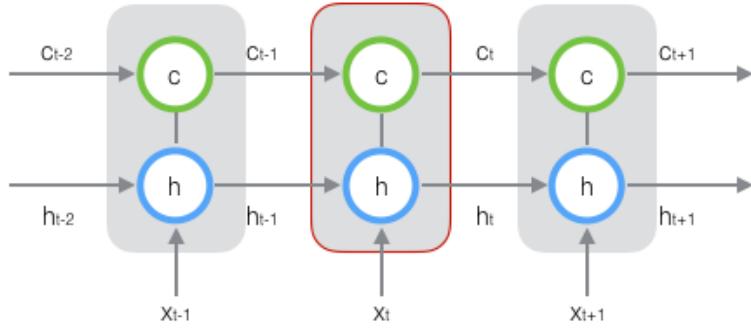


图 6.3: 梯度

图6.3仅仅是一个示意图，我们可以看出，在 $t$ 时刻，LSTM的输入有三个：当前时刻网络的输入值 $x_t$ 、上一时刻LSTM的输出值 $h_{t-1}$ 、以及上一时刻的单元状态 $c_{t-1}$ ；LSTM的输出有两个：当前时刻LSTM输出值 $h_t$ 、和当前时刻的单元状态 $c_t$ 。注意 $x$ 、 $h$ 、 $c$ 都是向量。

LSTM的关键，就是怎样控制长期状态 $c$ 。在这里，LSTM的思路是使用三个控制开关。第一个开关，负责控制继续保存长期状态 $c$ ；第二个开关，负责控制把即时状态输入到长期状态 $c$ ；第三个开关，负责控制是否把长期状态 $c$ 作为当前的LSTM的输出。三个开关的作用如图6.4所示：

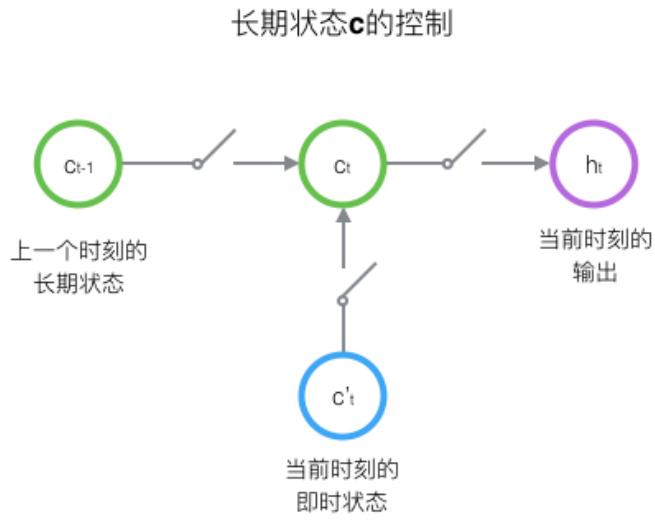


图 6.4: 梯度

接下来，我们要描述一下，输出 $h$ 和单元状态 $c$ 的具体计算方法。

## 6.2 长短时记忆网络的前向计算

前面描述的开关是怎样在算法中实现的呢？这就用到了门 (gate) 的概念。门实际上就是一层全连接层，它的输入是一个向量，输出是一个0到1之间的实数向量。假设 $W$

是门的权重向量， $b$  是偏置项，那么门可以表示为：

$$g(x) = \sigma(Wx + b)$$

门的使用，就是用门的输出向量按元素乘以我们需要控制的那个向量。因为门的输出是 0 到 1 之间的实数向量，那么，当门输出为 0 时，任何向量与之相乘都会得到 0 向量，这就相当于啥都不能通过；输出为 1 时，任何向量与之相乘都不会有任何改变，这就相当于啥都可以通过。因为  $\sigma$ （也就是 sigmoid 函数）的值域是 (0,1)，所以门的状态都是半开半闭的。

LSTM 用两个门来控制单元状态  $c$  的内容，一个是遗忘门（**forget gate**），它决定了上一时刻的单元状态  $c_{t-1}$  有多少保留到当前时刻  $c_t$ ；另一个是输入门（**input gate**），它决定了当前时刻网络的输入  $x_t$  有多少保存到单元状态  $c_t$ 。LSTM 用输出门（**output gate**）来控制单元状态  $c_t$  有多少输出到 LSTM 的当前输出值  $h_t$ 。

我们先来看一下遗忘门：

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad (6.1)$$

上式中， $W_f$  是遗忘门的权重矩阵， $[h_{t-1}, x_t]$  表示把两个向量连接成一个更长的向量， $b_f$  是遗忘门的偏置项， $\sigma$  是 sigmoid 函数。如果输入的维度是  $d_x$ ，隐藏层的维度是  $d_h$ ，单元状态的维度是  $d_c$ （通常  $d_c = d_h$ ），则遗忘门的权重矩阵  $W_f$  维度是  $d_c \times (d_h + d_x)$ 。事实上，权重矩阵  $W_f$  都是两个矩阵拼接而成的：一个是  $W_{fh}$ ，它对应着输入项  $h_{t-1}$ ，其维度为  $d_c \times d_h$ ；一个是  $W_{fx}$ ，它对应着输入项  $x_t$ ，其维度为  $d_c \times d_x$ 。 $W_f$  可以写为：

$$\begin{bmatrix} W_f \\ x_t \end{bmatrix} \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} = \begin{bmatrix} W_{fh} & W_{fx} \end{bmatrix} \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} \\ = W_{fh}h_{t-1} + W_{fx}x_t$$

图6.5显示了遗忘门的计算：

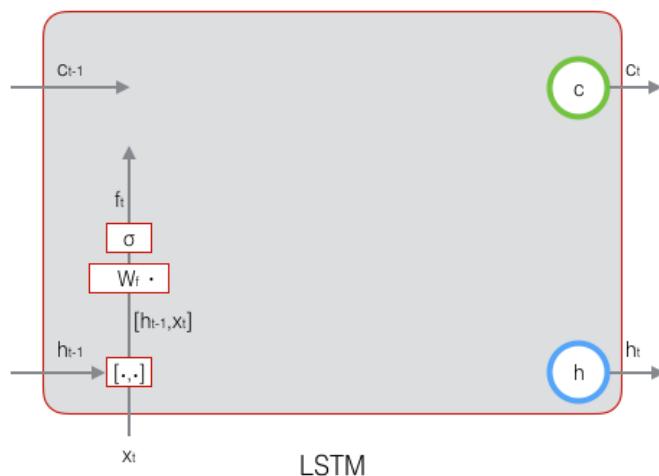


图 6.5: 遗忘门

接下来看看输入门：

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad (6.2)$$

上式中,  $W_i$  是输入门的权重矩阵,  $b_i$  是输入门的偏置项。图6.6表示了输入门的计算:

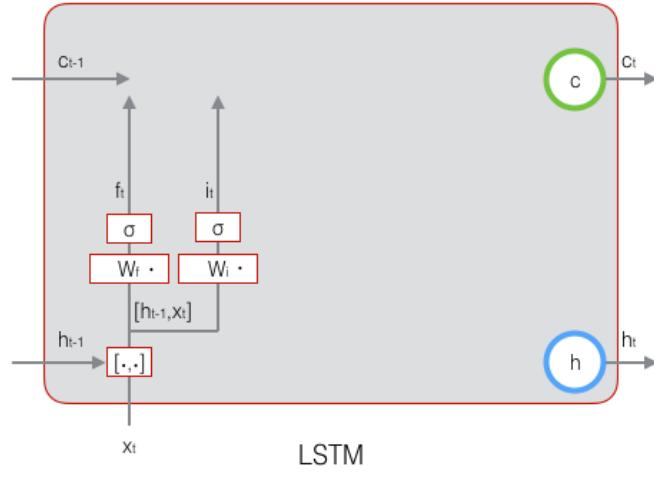
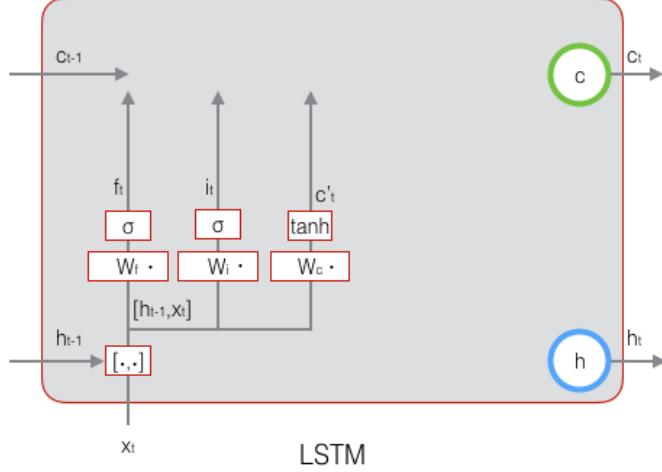


图 6.6: 输入门

接下来, 我们计算用于描述当前输入的单元状态  $\tilde{c}_t$ , 它是根据上一次的输出和本次输入来计算的:

$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \quad (6.3)$$

图6.7是  $\tilde{c}_t$  的计算: 现在, 我们计算当前时刻的单元状态  $c_t$ 。它是由上一次的单元状

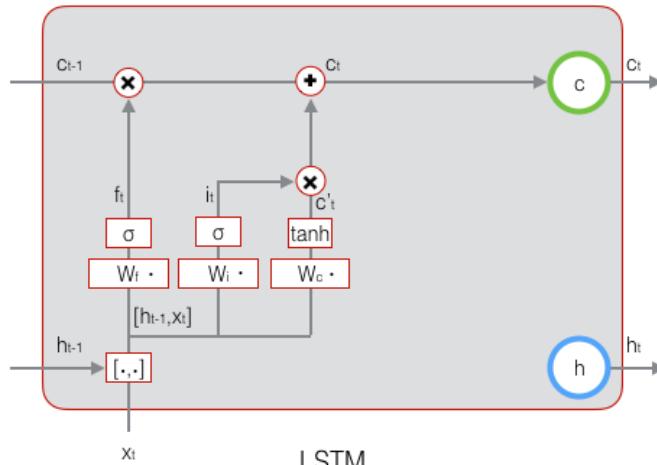
图 6.7:  $\tilde{c}_t$  的计算

态  $c_{t-1}$  按元素乘以遗忘门  $f_t$ , 再用当前输入的单元状态  $\tilde{c}_t$  按元素乘以输入门  $i_t$ , 再将两个积加和产生的:

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \quad (6.4)$$

符号  $\circ$  表示按元素乘。图6.8是  $c_t$  的计算:

这样, 我们就把 LSTM 关于当前的记忆  $\tilde{c}_t$  和长期的记忆  $c_{t-1}$  组合在一起, 形成了新的单元状态  $c_t$ 。由于遗忘门的控制, 它可以保存很久很久之前的信息, 由于输入门的控制, 它又可以避免当前无关紧要的内容进入记忆。下面, 我们要看看输出门, 它控制了

图 6.8:  $c_t$  的计算

长期记忆对当前输出的影响:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad (6.5)$$

图6.9表示输出门的计算:

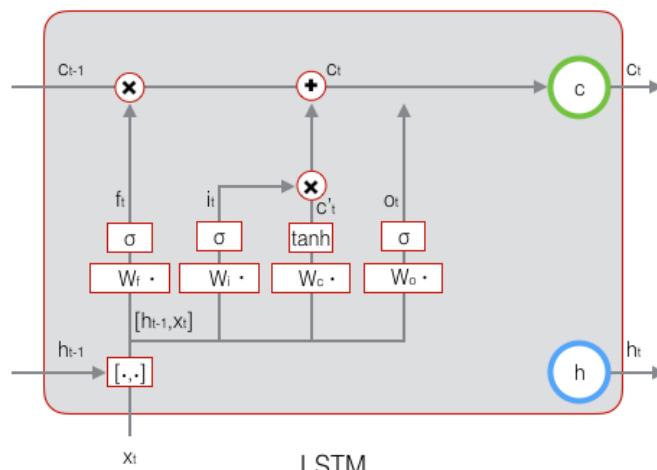


图 6.9: 输出门

LSTM 最终的输出，是由输出门和单元状态共同确定的:

$$h_t = o_t \circ \tanh(c_t) \quad (6.6)$$

图6.10表示 LSTM 最终输出的计算。

公式6.1到公式6.6就是 LSTM 前向计算的全部公式。至此，我们就把 LSTM 前向计算讲完了。

### 6.3 长短时记忆网络的训练

熟悉我们这个系列文章的同学都清楚，训练部分往往比前向计算部分复杂多了。LSTM 的前向计算都这么复杂，那么，可想而知，它的训练算法一定是非常非常复杂的。现在只有做几次深呼吸，再一头扎进公式海洋吧。

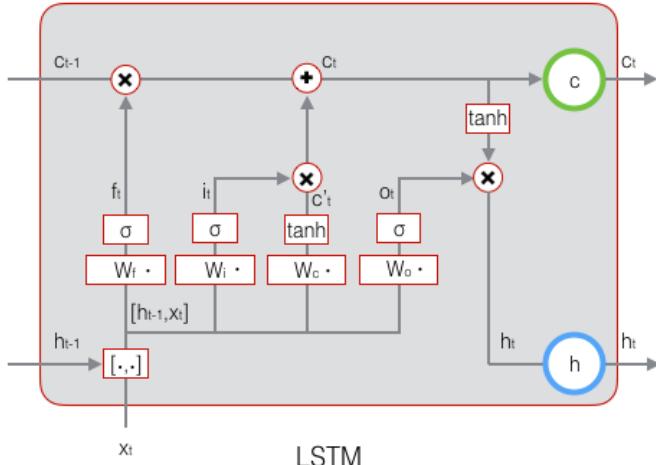


图 6.10: LSTM 最终输出

### 6.3.1 LSTM 训练算法框架

LSTM 的训练算法仍然是反向传播算法，对于这个算法，我们已经非常熟悉了。主要有下面三个步骤：

1. 前向计算每个神经元的输出值，对于 LSTM 来说，即  $f_t$ 、 $i_t$ 、 $c_t$ 、 $o_t$ 、 $h_t$  五个向量的值。计算方法已经在上一节中描述过了。
2. 反向计算每个神经元的误差项  $\delta$  值。与循环神经网络一样，LSTM 误差项的反向传播也是包括两个方向：一个是沿时间的反向传播，即从当前  $t$  时刻开始，计算每个时刻的误差项；一个是在将误差项向上一层传播。
3. 根据相应的误差项，计算每个权重的梯度。

### 6.3.2 关于公式和符号的说明

首先，我们对推导中用到的一些公式、符号做一下必要的说明。

接下来的推导中，我们设定 gate 的激活函数为 sigmoid 函数，输出的激活函数为 tanh 函数。他们的导数分别为：

$$\begin{aligned}\sigma(z) &= y = \frac{1}{1 + e^{-z}} \\ \sigma'(z) &= y(1 - y) \\ \tanh(z) &= y = \frac{e^z - e^{-z}}{e^z + e^{-z}} \\ \tanh'(z) &= 1 - y^2\end{aligned}$$

从上面可以看出，sigmoid 和 tanh 函数的导数都是原函数的函数。这样，我们一旦计算原函数的值，就可以用它来计算出导数的值。

LSTM 需要学习的参数共有 8 组，分别是：遗忘门的权重矩阵  $W_f$  和偏置项  $b_f$ 、输入门的权重矩阵  $W_i$  和偏置项  $b_i$ 、输出门的权重矩阵  $W_o$  和偏置项  $b_o$ ，以及计算单元状态的权重矩阵  $W_c$  和偏置项  $b_c$ 。因为权重矩阵的两部分在反向传播中使用不同的公式，因此在后续的推导中，权重矩阵  $W_f$ 、 $W_i$ 、 $W_c$ 、 $W_o$  都将被写为分开的两个矩阵： $W_{fh}$ 、 $W_{fx}$ 、

$W_{ih}$ 、 $W_{ix}$ 、 $W_{oh}$ 、 $W_{ox}$ 、 $W_{ch}$ 、 $W_{cx}$ 。

我们解释一下按元素乘 $\circ$ 符号。当 $\circ$ 作用于两个向量时，运算如下：

$$a \circ b = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \dots \\ a_n \end{bmatrix} \circ \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \dots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 b_1 \\ a_2 b_2 \\ a_3 b_3 \\ \dots \\ a_n b_n \end{bmatrix}$$

当 $\circ$ 作用于一个向量和一个矩阵时，运算如下：

$$\begin{aligned} a \circ X &= \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \dots \\ a_n \end{bmatrix} \circ \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ x_{31} & x_{32} & x_{33} & \dots & x_{3n} \\ \dots & & & \dots & \\ x_{n1} & x_{n2} & x_{n3} & \dots & x_{nn} \end{bmatrix} \\ &= \begin{bmatrix} a_1 x_{11} & a_1 x_{12} & a_1 x_{13} & \dots & a_1 x_{1n} \\ a_2 x_{21} & a_2 x_{22} & a_2 x_{23} & \dots & a_2 x_{2n} \\ a_3 x_{31} & a_3 x_{32} & a_3 x_{33} & \dots & a_3 x_{3n} \\ \dots & & & \dots & \\ a_n x_{n1} & a_n x_{n2} & a_n x_{n3} & \dots & a_n x_{nn} \end{bmatrix} \end{aligned}$$

当 $\circ$ 作用于两个矩阵时，两个矩阵对应位置的元素相乘。按元素乘可以在某些情况下简化矩阵和向量运算。例如，当一个对角矩阵右乘一个矩阵时，相当于用对角矩阵的对角线组成的向量按元素乘那个矩阵：

$$diag[a]X = a \circ X$$

当一个行向量右乘一个对角矩阵时，相当于这个行向量按元素乘那个矩阵对角线组成的向量：

$$a^T diag[b] = a \circ b$$

上面这两点，在我们后续推导中会多次用到。

在 $t$ 时刻，LSTM 的输出值为 $h_t$ 。我们定义 $t$ 时刻的误差项 $\delta_t$ 为：

$$\delta_t \stackrel{\text{def}}{=} \frac{\partial E}{\partial h_t}$$

注意，和前面几篇文章不同，我们这里假设误差项是损失函数对输出值的导数，而不是对加权输入 $net_t^l$ 的导数。因为LSTM有四个加权输入，分别对应 $f_t$ 、 $i_t$ 、 $c_t$ 、 $o_t$ ，我们希望往上一层传递一个误差项而不是四个。但我们仍然需要定义出这四个加权输入，以及他们对应的误差项。

$$\begin{aligned}
net_{f,t} &= W_f[h_{t-1}, x_t] + b_f = W_{fh}h_{t-1} + W_{fx}x_t + b_f \\
net_{i,t} &= W_i[h_{t-1}, x_t] + b_i = W_{ih}h_{t-1} + W_{ix}x_t + b_i \\
net_{\tilde{c},t} &= W_c[h_{t-1}, x_t] + b_c = W_{ch}h_{t-1} + W_{cx}x_t + b_c \\
net_{o,t} &= W_o[h_{t-1}, x_t] + b_o = W_{oh}h_{t-1} + W_{ox}x_t + b_o \\
\delta_{f,t} &\stackrel{\text{def}}{=} \frac{\partial E}{\partial net_{f,t}}, \delta_{i,t} \stackrel{\text{def}}{=} \frac{\partial E}{\partial net_{i,t}}, \delta_{\tilde{c},t} \stackrel{\text{def}}{=} \frac{\partial E}{\partial net_{\tilde{c},t}}, \delta_{o,t} \stackrel{\text{def}}{=} \frac{\partial E}{\partial net_{o,t}}
\end{aligned}$$

### 6.3.3 误差项沿时间的反向传递

沿时间反向传递误差项，就是要计算出 t-1 时刻的误差项  $\delta_{t-1}$ 。

$$\delta_{t-1}^T = \frac{\partial E}{\partial h_{t-1}} = \frac{\partial E}{\partial h_t} \frac{\partial h_t}{\partial h_{t-1}} = \delta_t^T \frac{\partial h_t}{\partial h_{t-1}}$$

我们知道， $\frac{\partial h_t}{\partial h_{t-1}}$  是一个 Jacobian 矩阵。如果隐藏层  $h$  的维度是 N 的话，那么它就是一个  $N \times N$  矩阵。为了求出它，我们列出  $h_t$  的计算公式，即前面的公式6.4和公式6.6：

$$h_t = o_t \circ \tanh(c_t)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$

显然， $o_t$ 、 $f_t$ 、 $i_t$ 、 $\tilde{c}_t$  都是  $h_{t-1}$  的函数，那么，利用全导数公式可得：

$$\begin{aligned}
\delta_t^T \frac{\partial h_t}{\partial h_{t-1}} &= \delta_t^T \frac{\partial h_t}{\partial o_t} \frac{\partial o_t}{\partial net_{o,t}} \frac{\partial net_{o,t}}{\partial h_{t-1}} + \delta_t^T \frac{\partial h_t}{\partial c_t} \frac{\partial c_t}{\partial f_t} \frac{\partial f_t}{\partial net_{f,t}} \frac{\partial net_{f,t}}{\partial h_{t-1}} \\
&\quad + \delta_t^T \frac{\partial h_t}{\partial c_t} \frac{\partial c_t}{\partial i_t} \frac{\partial i_t}{\partial net_{i,t}} \frac{\partial net_{i,t}}{\partial h_{t-1}} + \delta_t^T \frac{\partial h_t}{\partial c_t} \frac{\partial c_t}{\partial \tilde{c}_t} \frac{\partial \tilde{c}_t}{\partial net_{\tilde{c},t}} \frac{\partial net_{\tilde{c},t}}{\partial h_{t-1}} \\
&= \delta_{o,t}^T \frac{\partial net_{o,t}}{\partial h_{t-1}} + \delta_{f,t}^T \frac{\partial net_{f,t}}{\partial h_{t-1}} + \delta_{i,t}^T \frac{\partial net_{i,t}}{\partial h_{t-1}} + \delta_{\tilde{c},t}^T \frac{\partial net_{\tilde{c},t}}{\partial h_{t-1}} \tag{6.7}
\end{aligned}$$

下面，我们要把公式6.7中的每个偏导数都求出来。根据公式6.6，我们可以求出：

$$\frac{\partial h_t}{\partial o_t} = diag[\tanh(c_t)]$$

$$\frac{\partial h_t}{\partial c_t} = diag[o_t \circ (1 - \tanh(c_t)^2)]$$

根据公式6.4，我们可以求出：

$$\frac{\partial c_t}{\partial f_t} = diag[c_{t-1}], \quad \frac{\partial c_t}{\partial i_t} = diag[\tilde{c}_t], \quad \frac{\partial c_t}{\partial \tilde{c}_t} = diag[i_t]$$

因为：

$$o_t = \sigma(net_{o,t}), \quad net_{o,t} = W_{oh}h_{t-1} + W_{ox}x_t + b_o$$

$$f_t = \sigma(net_{f,t}), \quad net_{f,t} = W_{fh}h_{t-1} + W_{fx}x_t + b_f$$

$$i_t = \sigma(net_{i,t}), \quad net_{i,t} = W_{ih}h_{t-1} + W_{ix}x_t + b_i$$

$$\tilde{c}_t = \tanh(net_{\tilde{c},t}), \quad net_{\tilde{c},t} = W_{ch}h_{t-1} + W_{cx}x_t + b_c$$

我们很容易得出：

$$\begin{aligned}\frac{\partial o_t}{\partial net_{o,t}} &= diag[o_t \circ (1 - o_t)], & \frac{\partial net_{o,t}}{\partial h_{t-1}} &= W_{oh} \\ \frac{\partial f_t}{\partial net_{f,t}} &= diag[f_t \circ (1 - f_t)], & \frac{\partial net_{f,t}}{\partial h_{t-1}} &= W_{fh} \\ \frac{\partial i_t}{\partial net_{i,t}} &= diag[i_t \circ (1 - i_t)], & \frac{\partial net_{i,t}}{\partial h_{t-1}} &= W_{ih} \\ \frac{\partial \tilde{c}_t}{\partial net_{\tilde{c},t}} &= diag[1 - \tilde{c}_t^2], & \frac{\partial net_{\tilde{c},t}}{\partial h_{t-1}} &= W_{ch}\end{aligned}$$

将上述偏导数带入到公式6.7，我们得到：

$$\begin{aligned}\delta_{t-1} &= \delta_{o,t}^T \frac{\partial net_{o,t}}{\partial h_{t-1}} + \delta_{f,t}^T \frac{\partial net_{f,t}}{\partial h_{t-1}} + \delta_{i,t}^T \frac{\partial net_{i,t}}{\partial h_{t-1}} + \delta_{\tilde{c},t}^T \frac{\partial net_{\tilde{c},t}}{\partial h_{t-1}} \\ &= \delta_{o,t}^T W_{oh} + \delta_{f,t}^T W_{fh} + \delta_{i,t}^T W_{ih} + \delta_{\tilde{c},t}^T W_{ch}\end{aligned}\quad (6.8)$$

根据  $\delta_{o,t}$ 、 $\delta_{f,t}$ 、 $\delta_{i,t}$ 、 $\delta_{\tilde{c},t}$  的定义，可知：

$$\delta_{o,t}^T = \delta_t^T \circ \tanh(c_t) \circ o_t \circ (1 - o_t) \quad (6.9)$$

$$\delta_{f,t}^T = \delta_t^T \circ o_t \circ (1 - \tanh(c_t)^2) \circ c_{t-1} \circ f_t \circ (1 - f_t) \quad (6.10)$$

$$\delta_{i,t}^T = \delta_t^T \circ o_t \circ (1 - \tanh(c_t)^2) \circ \tilde{c}_t \circ i_t \circ (1 - i_t) \quad (6.11)$$

$$\delta_{\tilde{c},t}^T = \delta_t^T \circ o_t \circ (1 - \tanh(c_t)^2) \circ i_t \circ (1 - \tilde{c}_t^2) \quad (6.12)$$

公式6.8到公式6.12就是将误差沿时间反向传播一个时刻的公式。有了它，我们可以写出将误差项向前传递到任意 k 时刻的公式：

$$\delta_k^T = \prod_{j=k}^{t-1} \delta_{o,j}^T W_{oh} + \delta_{f,j}^T W_{fh} + \delta_{i,j}^T W_{ih} + \delta_{\tilde{c},j}^T W_{ch} \quad (6.13)$$

#### 6.3.4 将误差项传递到上一层

我们假设当前为第 1 层，定义 1-1 层的误差项是误差函数对 1-1 层加权输入的导数，即：

$$\delta_t^{l-1} \stackrel{def}{=} \frac{\partial E}{\partial net_t^{l-1}}$$

本次 LSTM 的输入  $x_t$  由下面的公式计算：

$$x_t^l = f^{l-1}(net_t^{l-1})$$

上式中， $f^{l-1}$  表示第 1-1 层的激活函数。

因为  $net_{f,t}^l$ 、 $net_{i,t}^l$ 、 $net_{\tilde{c},t}^l$ 、 $net_{o,t}^l$  都是  $x_t$  的函数， $x_t$  又是  $net_t^{l-1}$  的函数，因此，要求数出 E 对  $net_t^{l-1}$  的导数，就需要使用全导数公式：

$$\begin{aligned}
\frac{\partial E}{\partial net_t^{l-1}} &= \frac{\partial E}{\partial net_{f,t}^l} \frac{\partial net_{f,t}^l}{\partial x_t^l} \frac{\partial x_t^l}{\partial net_t^{l-1}} + \frac{\partial E}{\partial net_{i,t}^l} \frac{\partial net_{i,t}^l}{\partial x_t^l} \frac{\partial x_t^l}{\partial net_t^{l-1}} \\
&\quad + \frac{\partial E}{\partial net_{\tilde{c},t}^l} \frac{\partial net_{\tilde{c},t}^l}{\partial x_t^l} \frac{\partial x_t^l}{\partial net_t^{l-1}} + \frac{\partial E}{\partial net_{o,t}^l} \frac{\partial net_{o,t}^l}{\partial x_t^l} \frac{\partial x_t^l}{\partial net_t^{l-1}} \\
&= \delta_{f,t}^T W_{fx} \circ f'(net_t^{l-1}) + \delta_{i,t}^T W_{ix} \circ f'(net_t^{l-1}) + \delta_{\tilde{c},t}^T W_{cx} \circ f'(net_t^{l-1}) + \delta_{o,t}^T W_{ox} \circ f'(net_t^{l-1}) \\
&= (\delta_{f,t}^T W_{fx} + \delta_{i,t}^T W_{ix} + \delta_{\tilde{c},t}^T W_{cx} + \delta_{o,t}^T W_{ox}) \circ f'(net_t^{l-1})
\end{aligned} \tag{6.14}$$

公式6.14就是将误差传递到上一层的公式。

### 6.3.5 权重梯度的计算

对于  $W_{fh}$ 、 $W_{ih}$ 、 $W_{ch}$ 、 $W_{oh}$  的权重梯度，我们知道它的梯度是各个时刻梯度之和（证明过程请参考第5循环神经网络），我们首先求出它们在  $t$  时刻的梯度，然后再求出他们最终的梯度。

我们已经求得了误差项  $\delta_{o,t}$ 、 $\delta_{f,t}$ 、 $\delta_{i,t}$ 、 $\delta_{\tilde{c},t}$ ，很容易求出  $t$  时刻的  $W_{oh}$ 、 $W_{ih}$ 、 $W_{fh}$ 、 $W_{ch}$  的梯度：

$$\begin{aligned}
\frac{\partial E}{\partial W_{oh,t}} &= \frac{\partial E}{\partial net_{o,t}} \frac{\partial net_{o,t}}{\partial W_{oh,t}} = \delta_{o,t} h_{t-1}^T \\
\frac{\partial E}{\partial W_{fh,t}} &= \frac{\partial E}{\partial net_{f,t}} \frac{\partial net_{f,t}}{\partial W_{fh,t}} = \delta_{f,t} h_{t-1}^T \\
\frac{\partial E}{\partial W_{ih,t}} &= \frac{\partial E}{\partial net_{i,t}} \frac{\partial net_{i,t}}{\partial W_{ih,t}} = \delta_{i,t} h_{t-1}^T \\
\frac{\partial E}{\partial W_{ch,t}} &= \frac{\partial E}{\partial net_{\tilde{c},t}} \frac{\partial net_{\tilde{c},t}}{\partial W_{ch,t}} = \delta_{\tilde{c},t} h_{t-1}^T
\end{aligned}$$

将各个时刻的梯度加在一起，就能得到最终的梯度：

$$\begin{aligned}
\frac{\partial E}{\partial W_{oh}} &= \sum_{j=1}^t \delta_{o,j} h_{j-1}^T \\
\frac{\partial E}{\partial W_{fh}} &= \sum_{j=1}^t \delta_{f,j} h_{j-1}^T \\
\frac{\partial E}{\partial W_{ih}} &= \sum_{j=1}^t \delta_{i,j} h_{j-1}^T \\
\frac{\partial E}{\partial W_{ch}} &= \sum_{j=1}^t \delta_{\tilde{c},j} h_{j-1}^T
\end{aligned}$$

对于偏置项  $b_f$ 、 $b_i$ 、 $b_c$ 、 $b_o$  的梯度，也是将各个时刻的梯度加在一起。下面是各个

时刻的偏置项梯度：

$$\begin{aligned}\frac{\partial E}{\partial b_{o,t}} &= \frac{\partial E}{\partial net_{o,t}} \frac{\partial net_{o,t}}{\partial b_{o,t}} = \delta_{o,t} \\ \frac{\partial E}{\partial b_{f,t}} &= \frac{\partial E}{\partial net_{f,t}} \frac{\partial net_{f,t}}{\partial b_{f,t}} = \delta_{f,t} \\ \frac{\partial E}{\partial b_{i,t}} &= \frac{\partial E}{\partial net_{i,t}} \frac{\partial net_{i,t}}{\partial b_{i,t}} = \delta_{i,t} \\ \frac{\partial E}{\partial b_{c,t}} &= \frac{\partial E}{\partial net_{\tilde{c},t}} \frac{\partial net_{\tilde{c},t}}{\partial b_{c,t}} = \delta_{\tilde{c},t}\end{aligned}$$

下面是最终的偏置项梯度，即将各个时刻的偏置项梯度加在一起：

$$\frac{\partial E}{\partial b_o} = \sum_{j=1}^t \delta_{o,j}, \frac{\partial E}{\partial b_i} = \sum_{j=1}^t \delta_{i,j}, \frac{\partial E}{\partial b_f} = \sum_{j=1}^t \delta_{f,j}, \frac{\partial E}{\partial b_c} = \sum_{j=1}^t \delta_{\tilde{c},j}$$

对于  $W_{fx}$ 、 $W_{ix}$ 、 $W_{cx}$ 、 $W_{ox}$  的权重梯度，只需要根据相应的误差项直接计算即可：

$$\begin{aligned}\frac{\partial E}{\partial W_{ox}} &= \frac{\partial E}{\partial net_{o,t}} \frac{\partial net_{o,t}}{\partial W_{ox}} = \delta_{o,t} x_t^T \\ \frac{\partial E}{\partial W_{fx}} &= \frac{\partial E}{\partial net_{f,t}} \frac{\partial net_{f,t}}{\partial W_{fx}} = \delta_{f,t} x_t^T \\ \frac{\partial E}{\partial W_{ix}} &= \frac{\partial E}{\partial net_{i,t}} \frac{\partial net_{i,t}}{\partial W_{ix}} = \delta_{i,t} x_t^T \\ \frac{\partial E}{\partial W_{cx}} &= \frac{\partial E}{\partial net_{\tilde{c},t}} \frac{\partial net_{\tilde{c},t}}{\partial W_{cx}} = \delta_{\tilde{c},t} x_t^T\end{aligned}$$

以上就是 LSTM 的训练算法的全部公式。因为这里面存在很多重复的模式，仔细看看，会发觉并不是太复杂。

当然，LSTM 存在着相当多的变体，读者可以在互联网上找到很多资料。因为大家已经熟悉了基本 LSTM 的算法，因此理解这些变体比较容易，因此本文就不再赘述了。

## 6.4 编程实战：长短时记忆网络的实现



**注意** 完整代码请参考 GitHub: [https://github.com/hanbt/learn\\_dl/blob/master/lstm.py](https://github.com/hanbt/learn_dl/blob/master/lstm.py) (python2.7)

在下面的实现中，LSTMLayer 的参数包括输入维度、输出维度、隐藏层维度，单元状态维度等于隐藏层维度。gate 的激活函数为 sigmoid 函数，输出的激活函数为 tanh。

### 6.4.1 激活函数的实现

我们先实现两个激活函数：sigmoid 和 tanh。

```
1 class SigmoidActivator(object):
2     def forward(self, weighted_input):
3         return 1.0 / (1.0 + np.exp(-weighted_input))
4     def backward(self, output):
5         return output * (1 - output)
6 class TanhActivator(object):
```

```
7     def forward(self, weighted_input):
8         return 2.0 / (1.0 + np.exp(-2 * weighted_input)) - 1.0
9     def backward(self, output):
10        return 1 - output * output
```

### 6.4.2 LSTM 初始化

和前两篇文章代码架构一样，我们把 LSTM 的实现放在 LstmLayer 类中。

根据 LSTM 前向计算和反向传播算法，我们需要初始化一系列矩阵和向量。这些矩阵和向量有两类用途，一类是用于保存模型参数，例如  $W_f$ 、 $W_i$ 、 $W_o$ 、 $W_c$ 、 $b_f$ 、 $b_i$ 、 $b_o$ 、 $b_c$ ；另一类是保存各种中间计算结果，以便于反向传播算法使用，它们包括  $h_t$ 、 $f_t$ 、 $i_t$ 、 $o_t$ 、 $c_t$ 、 $\tilde{c}_t$ 、 $\delta_t$ 、 $\delta_{f,t}$ 、 $\delta_{i,t}$ 、 $\delta_{o,t}$ 、 $\delta_{\tilde{c},t}$ ，以及各个权重对应的梯度。

在构造函数的初始化中，只初始化了与 forward 计算相关的变量，与 backward 相关的变量没有初始化。这是因为构造 LSTM 对象的时候，我们还不知道它未来是用于训练（既有 forward 又有 backward）还是推理（只有 forward）。

```
1 class LstmLayer(object):
2     def __init__(self, input_width, state_width,
3                  learning_rate):
4         self.input_width = input_width
5         self.state_width = state_width
6         self.learning_rate = learning_rate
7         # 门的激活函数
8         self.gate_activator = SigmoidActivator()
9         # 输出的激活函数
10        self.output_activator = TanhActivator()
11        # 当前时刻初始化为 t0
12        self.times = 0
13        # 各个时刻的单元状态向量 c
14        self.c_list = self.init_state_vec()
15        # 各个时刻的输出向量 h
16        self.h_list = self.init_state_vec()
17        # 各个时刻的遗忘门 f
18        self.f_list = self.init_state_vec()
19        # 各个时刻的输入门 i
20        self.i_list = self.init_state_vec()
21        # 各个时刻的输出门 o
22        self.o_list = self.init_state_vec()
23        # 各个时刻的即时状态 c~
24        self.ct_list = self.init_state_vec()
25        # 遗忘门权重矩阵 Wfh, Wfx, 偏置项 bf
26        self.Wfh, self.Wfx, self.bf =
27            self.init_weight_mat()
```



```

28     # 输入门权重矩阵  $W_{fh}$ ,  $W_{fx}$ , 偏置项  $bf$ 
29     self.Wih, self.Wix, self.bi = (
30         self.init_weight_mat())
31     # 输出门权重矩阵  $W_{fh}$ ,  $W_{fx}$ , 偏置项  $bf$ 
32     self.Woh, self.Wox, self.bo = (
33         self.init_weight_mat())
34     # 单元状态权重矩阵  $W_{fh}$ ,  $W_{fx}$ , 偏置项  $bf$ 
35     self.Wch, self.Wcx, self.bc = (
36         self.init_weight_mat())
37     def init_state_vec(self):
38         ...
39         初始化保存状态的向量
40         ...
41         state_vec_list = []
42         state_vec_list.append(np.zeros(
43             (self.state_width, 1)))
44         return state_vec_list
45     def init_weight_mat(self):
46         ...
47         初始化权重矩阵
48         ...
49         Wh = np.random.uniform(-1e-4, 1e-4,
50             (self.state_width, self.state_width))
51         Wx = np.random.uniform(-1e-4, 1e-4,
52             (self.state_width, self.input_width))
53         b = np.zeros((self.state_width, 1))
54         return Wh, Wx, b

```

### 6.4.3 前向计算的实现

`forward` 方法实现了 LSTM 的前向计算:

```

1     def forward(self, x):
2         ...
3         根据式1-式6进行前向计算
4         ...
5         self.times += 1
6         # 遗忘门
7         fg = self.calc_gate(x, self.Wfx, self.Wfh,
8             self.bf, self.gate_activator)
9         self.f_list.append(fg)
10        # 输入门
11        ig = self.calc_gate(x, self.Wix, self.Wih,
12            self.bi, self.gate_activator)

```

```

13         self.i_list.append(ig)
14         # 输出门
15         og = self.calc_gate(x, self.Wox, self.Woh,
16                               self.bo, self.gate_activator)
17         self.o_list.append(og)
18         # 即时状态
19         ct = self.calc_gate(x, self.Wcx, self.Wch,
20                               self.bc, self.output_activator)
21         self.ct_list.append(ct)
22         # 单元状态
23         c = fg * self.c_list[self.times - 1] + ig * ct
24         self.c_list.append(c)
25         # 输出
26         h = og * self.output_activator.forward(c)
27         self.h_list.append(h)
28     def calc_gate(self, x, Wx, Wh, b, activator):
29         ...
30         计算门
31         ...
32         h = self.h_list[self.times - 1] # 上次的LSTM输出
33         net = np.dot(Wh, h) + np.dot(Wx, x) + b
34         gate = activator.forward(net)
35         return gate

```

从上面的代码我们可以看到，门的计算都是相同的算法，而门和 $c_t$ 的计算仅仅是激活函数不同。因此我们提出了calc\_gate方法，这样减少了很多重复代码。

#### 6.4.4 反向传播算法的实现

backward方法实现了LSTM的反向传播算法。需要注意的是，与backward相关的内部状态变量是在调用backward方法之后才初始化的。这种延迟初始化的一个好处是，如果LSTM只是用来推理，那么就不需要初始化这些变量，节省了很多内存。

```

1     def backward(self, x, delta_h, activator):
2         ...
3         实现LSTM训练算法
4         ...
5         self.calc_delta(delta_h, activator)
6         self.calc_gradient(x)

```

算法主要分成两个部分，一部分使计算误差项：

```

1     def calc_delta(self, delta_h, activator):
2         # 初始化各个时刻的误差项
3         self.delta_h_list = self.init_delta() # 输出误差项
4         self.delta_o_list = self.init_delta() # 输出门误差项

```



```
48     delta_h_prev = (
49         np.dot(delta_o.transpose(), self.Woh) +
50         np.dot(delta_i.transpose(), self.Wih) +
51         np.dot(delta_f.transpose(), self.Wfh) +
52         np.dot(delta_ct.transpose(), self.Wch)
53     ).transpose()
54     # 保存全部 delta 值
55     self.delta_h_list[k-1] = delta_h_prev
56     self.delta_f_list[k] = delta_f
57     self.delta_i_list[k] = delta_i
58     self.delta_o_list[k] = delta_o
59     self.delta_ct_list[k] = delta_ct
```

另一部分是计算梯度：

```
1 def calc_gradient(self, x):
2     # 初始化遗忘门权重梯度矩阵和偏置项
3     self.Wfh_grad, self.Wfx_grad, self.bf_grad = (
4         self.init_weight_gradient_mat())
5     # 初始化输入门权重梯度矩阵和偏置项
6     self.Wih_grad, self.Wix_grad, self.bi_grad = (
7         self.init_weight_gradient_mat())
8     # 初始化输出门权重梯度矩阵和偏置项
9     self.Woh_grad, self.Wox_grad, self.bo_grad = (
10        self.init_weight_gradient_mat())
11    # 初始化单元状态权重梯度矩阵和偏置项
12    self.Wch_grad, self.Wcx_grad, self.bc_grad = (
13        self.init_weight_gradient_mat())
14    # 计算对上一次输出 h 的权重梯度
15    for t in range(self.times, 0, -1):
16        # 计算各个时刻的梯度
17        (Wfh_grad, bf_grad,
18         Wih_grad, bi_grad,
19         Woh_grad, bo_grad,
20         Wch_grad, bc_grad) = (
21             self.calc_gradient_t(t))
22        # 实际梯度是各时刻梯度之和
23        self.Wfh_grad += Wfh_grad
24        self.bf_grad += bf_grad
25        self.Wih_grad += Wih_grad
26        self.bi_grad += bi_grad
27        self.Woh_grad += Woh_grad
28        self.bo_grad += bo_grad
29        self.Wch_grad += Wch_grad
```

```
30         self.bc_grad += bc_grad
31         print '-----%d-----' % t
32         print Wfh_grad
33         print self.Wfh_grad
34     # 计算对本次输入x的权重梯度
35     xt = x.transpose()
36     self.Wfx_grad = np.dot(self.delta_f_list[-1], xt)
37     self.Wix_grad = np.dot(self.delta_i_list[-1], xt)
38     self.Wox_grad = np.dot(self.delta_o_list[-1], xt)
39     self.Wcx_grad = np.dot(self.delta_ct_list[-1], xt)
40     def init_weight_gradient_mat(self):
41         ...
42     初始化权重矩阵
43     ...
44     Wh_grad = np.zeros((self.state_width,
45                         self.state_width))
46     Wx_grad = np.zeros((self.state_width,
47                         self.input_width))
48     b_grad = np.zeros((self.state_width, 1))
49     return Wh_grad, Wx_grad, b_grad
50     def calc_gradient_t(self, t):
51         ...
52         计算每个时刻t权重的梯度
53         ...
54         h_prev = self.h_list[t-1].transpose()
55         Wfh_grad = np.dot(self.delta_f_list[t], h_prev)
56         bf_grad = self.delta_f_list[t]
57         Wih_grad = np.dot(self.delta_i_list[t], h_prev)
58         bi_grad = self.delta_f_list[t]
59         Woh_grad = np.dot(self.delta_o_list[t], h_prev)
60         bo_grad = self.delta_f_list[t]
61         Wch_grad = np.dot(self.delta_ct_list[t], h_prev)
62         bc_grad = self.delta_ct_list[t]
63         return Wfh_grad, bf_grad, Wih_grad, bi_grad, \
64                 Woh_grad, bo_grad, Wch_grad, bc_grad
```

#### 6.4.5 梯度下降算法的实现

下面是用梯度下降算法来更新权重:

```
1  def update(self):
2      ...
3      按照梯度下降，更新权重
4      ...
```

```

5     self.Wfh -= self.learning_rate * self.Whf_grad
6     self.Wfx -= self.learning_rate * self.Whx_grad
7     self.bf -= self.learning_rate * self.bf_grad
8     self.Wih -= self.learning_rate * self.Whi_grad
9     self.Wix -= self.learning_rate * self.Whi_grad
10    self.bi -= self.learning_rate * self.bi_grad
11    self.Woh -= self.learning_rate * self.Wof_grad
12    self.Wox -= self.learning_rate * self.Wox_grad
13    self.bo -= self.learning_rate * self.bo_grad
14    self.Wch -= self.learning_rate * self.Wcf_grad
15    self.Wcx -= self.learning_rate * self.Wcx_grad
16    self.bc -= self.learning_rate * self.bc_grad

```

#### 6.4.6 梯度检查的实现

和 RecurrentLayer 一样，为了支持梯度检查，我们需要支持重置内部状态：

```

1  def reset_state(self):
2      # 当前时刻初始化为 t0
3      self.times = 0
4      # 各个时刻的单元状态向量 c
5      self.c_list = self.init_state_vec()
6      # 各个时刻的输出向量 h
7      self.h_list = self.init_state_vec()
8      # 各个时刻的遗忘门 f
9      self.f_list = self.init_state_vec()
10     # 各个时刻的输入门 i
11     self.i_list = self.init_state_vec()
12     # 各个时刻的输出门 o
13     self.o_list = self.init_state_vec()
14     # 各个时刻的即时状态 c~
15     self.ct_list = self.init_state_vec()

```

最后，是梯度检查的代码：

```

1  def data_set():
2      x = [np.array([[1], [2], [3]]),
3            np.array([[2], [3], [4]])]
4      d = np.array([[1], [2]])
5      return x, d
6  def gradient_check():
7      ...
8      梯度检查
9      ...
10     # 设计一个误差函数，取所有节点输出项之和

```

```

11     error_function = lambda o: o.sum()
12     lstm = LstmLayer(3, 2, 1e-3)
13     # 计算forward值
14     x, d = data_set()
15     lstm.forward(x[0])
16     lstm.forward(x[1])
17     # 求取sensitivity map
18     sensitivity_array = np.ones(lstm.h_list[-1].shape,
19                                   dtype=np.float64)
20     # 计算梯度
21     lstm.backward(x[1], sensitivity_array, IdentityActivator())
22     # 检查梯度
23     epsilon = 10e-4
24     for i in range(lstm.Wfh.shape[0]):
25         for j in range(lstm.Wfh.shape[1]):
26             lstm.Wfh[i, j] += epsilon
27             lstm.reset_state()
28             lstm.forward(x[0])
29             lstm.forward(x[1])
30             err1 = error_function(lstm.h_list[-1])
31             lstm.Wfh[i, j] -= 2*epsilon
32             lstm.reset_state()
33             lstm.forward(x[0])
34             lstm.forward(x[1])
35             err2 = error_function(lstm.h_list[-1])
36             expect_grad = (err1 - err2) / (2 * epsilon)
37             lstm.Wfh[i, j] += epsilon
38             print 'weights(%d,%d): expected - actural %.4e - %.4e' %
39                 (i, j, expect_grad, lstm.Wfh_grad[i, j])
40     return lstm

```

我们只对  $W_{fh}$  做了检查，读者可以自行增加对其他梯度的检查。下面是某次梯度检查的结果：

```

weights(0,0): expected - actural 1.3908e-09 - 1.3909e-09
weights(0,1): expected - actural 1.4017e-09 - 1.4017e-09
weights(1,0): expected - actural 1.4017e-09 - 1.4017e-09
weights(1,1): expected - actural 1.4126e-09 - 1.4126e-09

```

## 6.5 GRU

前面我们讲了一种普通的 LSTM，事实上 LSTM 存在很多变体，许多论文中的 LSTM 都或多或少的不太一样。在众多的 LSTM 变体中，**GRU (Gated Recurrent Unit)** 也许是 最成功的一种。它对 LSTM 做了很多简化，同时却保持着和 LSTM 相同的效果。因此，

GRU 最近变得越来越流行。

GRU 对 LSTM 做了两个大改动：

1. 将输入门、遗忘门、输出门变为两个门：更新门（Update Gate） $z_t$  和重置门（Reset Gate） $r_t$ 。
2. 将单元状态与输出合并为一个状态： $h$ 。

GRU 的前向计算公式为：

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

$$\tilde{h}_t = \tanh(W \cdot [r_t \circ h_{t-1}, x_t])$$

$$h = (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h}_t$$

图6.11是 GRU 的示意图：

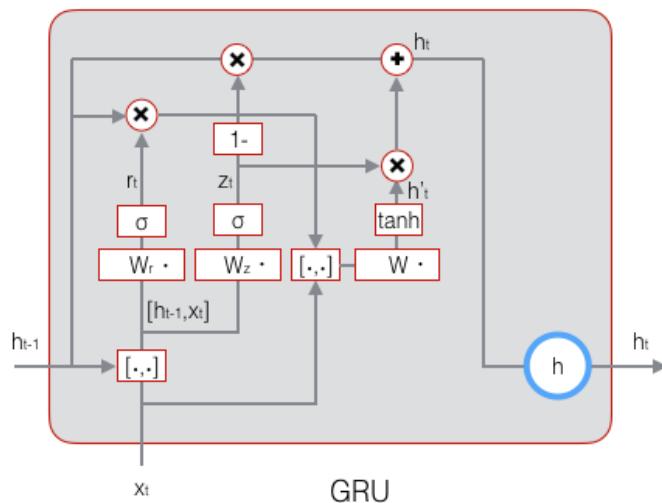


图 6.11: GRU

GRU 的训练算法比 LSTM 简单一些，留给读者自行推导，本文就不再赘述了。

## 6.6 小结

至此，LSTM---也许是结构最复杂的一类神经网络---就讲完了，相信拿下前几篇文章的读者们搞定这篇文章也不在话下吧！现在我们已经了解循环神经网络和它最流行的变体—**LSTM**，它们都可以用来处理序列。但是，有时候仅仅拥有处理序列的能力还不够，还需要处理比序列更为复杂的结构（比如树结构），这时候就需要用到另外一类网络：**递归神经网络 (Recursive Neural Network)**，巧合的是，它的缩写也是 **RNN**。在下一篇文章中，我们将介绍递归神经网络和它的训练算法。现在，漫长的烧脑暂告一段落，休息一下吧：)

# 第7章 循环神经网络

## 内容提要

- 递归神经网络是啥 7.1
- 递归神经网络的前向计算 7.2
- 递归神经网络的训练 7.3
- 误差项的传递 7.3.1
- 权重梯度的计算 7.3.2
- 权重更新 7.3.3
- 编程实战：递归神经网络的实现 7.4
- 递归神经网络的应用 7.5
- 自然语言和自然场景解析 7.5.1

在前面的文章中，我们介绍了循环神经网络，它可以用来处理包含序列结构的信息。然而，除此之外，信息往往还存在着诸如树结构、图结构等更复杂的结构。对于这种复杂的结构，循环神经网络就无能为力了。本文介绍一种更为强大、复杂的神经网络：递归神经网络 (**Recursive Neural Network, RNN**)，以及它的训练算法 **BPTS (Back Propagation Through Structure)**。顾名思义，递归神经网络（巧合的是，它的缩写和循环神经网络一样，也是 RNN）可以处理诸如树、图这样的递归结构。在文章的最后，我们将实现一个递归神经网络，并介绍它的几个应用场景。

## 7.1 递归神经网络是啥

因为神经网络的输入层单元个数是固定的，因此必须用循环或者递归的方式来处理长度可变的输入。循环神经网络实现了前者，通过将长度不定的输入分割为等长度的小块，然后再依次的输入到网络中，从而实现了神经网络对变长输入的处理。一个典型的例子是，当我们处理一句话的时候，我们可以把一句话看作是词组成的序列，然后，每次向循环神经网络输入一个词，如此循环直至整句话输入完毕，循环神经网络将产生对应的输出。如此，我们就能处理任意长度的句子了。如图7.1所示：

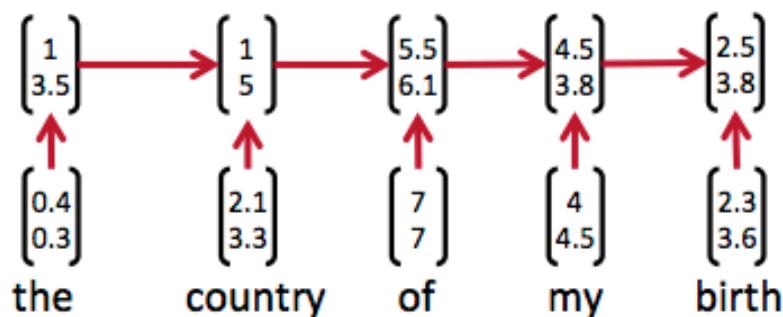


图 7.1：句子处理过程

然而，有时候把句子看做是词的序列是不够的，比如下面图7.2这句话『两个外语学院的学生』：

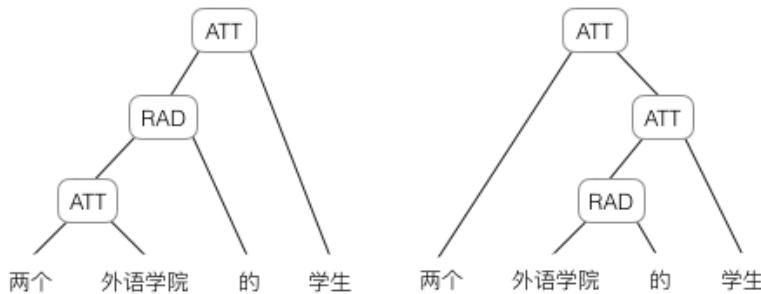


图 7.2: 两个外语学院的学生

图7.1显示了这句话的两个不同的语法解析树。可以看出来这句话有歧义，不同的语法解析树则对应了不同的意思。一个是『两个外语学院的/学生』，也就是学生可能有许多，但他们来自于两所外语学校；另一个是『两个/外语学院的学生』，也就是只有两个学生，他们是外语学院的。为了能够让模型区分出两个不同的意思，我们的模型必须能够按照树结构去处理信息，而不是序列，这就是递归神经网络的作用。当面对按照树/图结构处理信息更有效的任务时，递归神经网络通常都会获得不错的结果。

递归神经网络可以把一个树/图结构信息编码为一个向量，也就是把信息映射到一个语义向量空间中。这个语义向量空间满足某类性质，比如语义相似的向量距离更近。也就是说，如果两句话（尽管内容不同）它的意思是相似的，那么把它们分别编码后的两个向量的距离也相近；反之，如果两句话的意思截然不同，那么编码后向量的距离则很远。如图7.3所示：

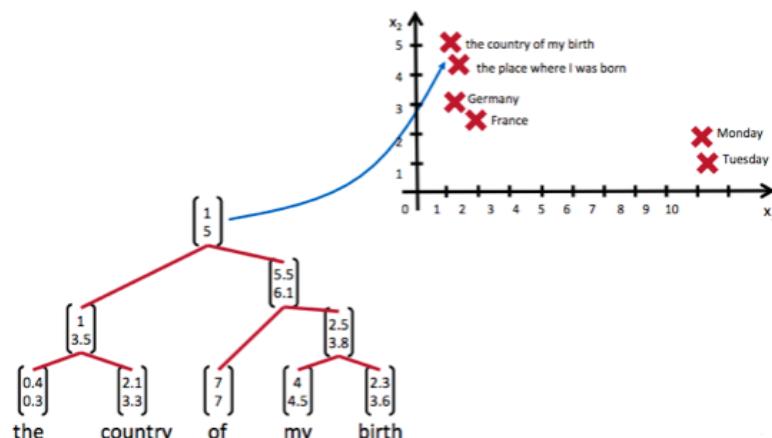


图 7.3: 编码后向量的距离

从图7.3我们可以看到，递归神经网络将所有的词、句都映射到一个2维向量空间中。句子『the country of my birth』和句子『the place where I was born』的意思是非常接近的，所以表示它们的两个向量在向量空间中的距离很近。另外两个词『Germany』和『France』因为表示的都是地点，它们的向量与上面两句话的向量的距离，就比另外两个表示时间的词『Monday』和『Tuesday』的向量的距离近得多。这样，通过向量的距离，就得到了一种语义的表示。

上图还显示了自然语言可组合的性质：词可以组成句、句可以组成段落、段落可以组成篇章，而更高层的语义取决于底层的语义以及它们的组合方式。递归神经网络是一

种表示学习，它可以将词、句、段、篇按照他们的语义映射到同一个向量空间中，也就是把可组合（树/图结构）的信息表示为一个个有意义的向量。比如上面这个例子，递归神经网络把句子“the country of my birth”表示为二维向量 [1,5]。有了这个『编码器』之后，我们就可以以这些有意义的向量为基础去完成更高级的任务（比如情感分析等）。如下图所示，递归神经网络在做情感分析时，可以比较好的处理否定句，这是胜过其他一些模型的：

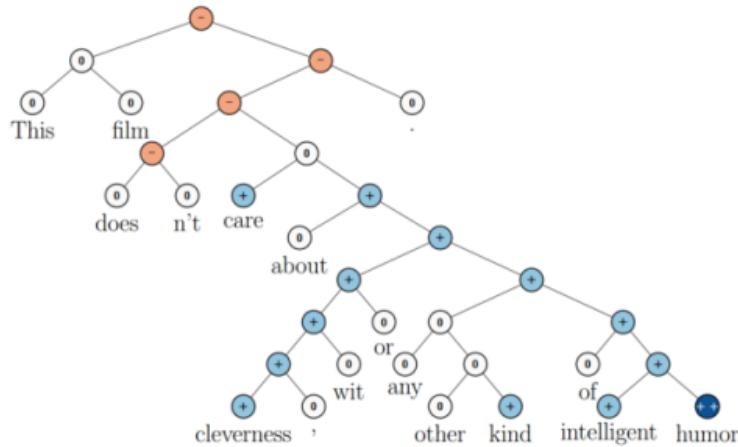


图 7.4: 递归神经网络

在图7.4中，蓝色表示正面评价，红色表示负面评价。每个节点是一个向量，这个向量表达了以它为根的子树的情感评价。比如“intelligent humor”是正面评价，而“care about cleverness wit or any other kind of intelligent humor”是中性评价。我们可以看到，模型能够正确的处理 doesn't 的含义，将正面评价转变为负面评价。

尽管递归神经网络具有更为强大的表示能力，但是在实际应用中并不太流行。其中一个主要原因是，递归神经网络的输入是树/图结构，而这种结构需要花费很多人工去标注。想象一下，如果我们用循环神经网络处理句子，那么我们可以直接把句子作为输入。然而，如果我们用递归神经网络处理句子，我们就必须把每个句子标注为语法解析树的形式，这无疑要花费非常大的精力。很多时候，相对于递归神经网络能够带来的性能提升，这个投入是不太划算的。

我们已经基本了解了递归神经网络是做什么用的，接下来，我们将探讨它的算法细节。

## 7.2 递归神经网络的前向计算

接下来，我们详细介绍一下递归神经网络是如何处理树/图结构的信息的。在这里，我们以处理树型信息为例进行介绍。

递归神经网络的输入是两个子节点（也可以是多个），输出就是将这两个子节点编码后产生的父节点，父节点的维度和每个子节点是相同的。如图7.5所示：

$c_1$  和  $c_2$  分别是表示两个子节点的向量， $p$  是表示父节点的向量。子节点和父节点组成一个全连接神经网络，也就是子节点的每个神经元都和父节点的每个神经元两两相连。

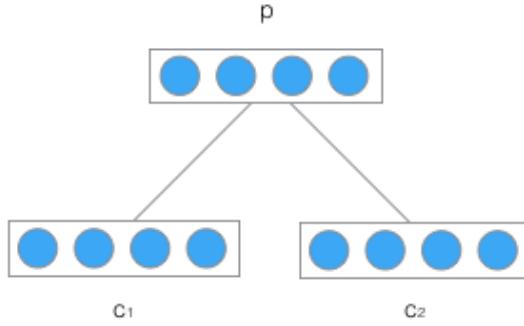


图 7.5: 神经元两两相连

我们用矩阵  $W$  表示这些连接上的权重，它的维度将是  $d \times 2d$ ，其中， $d$  表示每个节点的维度。父节点的计算公式可以写成：

$$p = \tanh(W \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} + b) \quad (7.1)$$

在上式中， $\tanh$  是激活函数（当然也可以用其它的激活函数）， $b$  是偏置项，它也是一个维度为  $d$  的向量。如果读过前面的文章，相信大家已经非常熟悉这些计算了，在此不做过多的解释了。

然后，我们把产生的父节点的向量和其他子节点的向量再次作为网络的输入，再次产生它们的父节点。如此递归下去，直至整棵树处理完毕。最终，我们将得到根节点的向量，我们可以认为它是对整棵树的表示，这样我们就实现了把树映射为一个向量。在图7.6中，我们使用递归神经网络处理一棵树，最终得到的向量  $p_3$ ，就是对整棵树的表示：

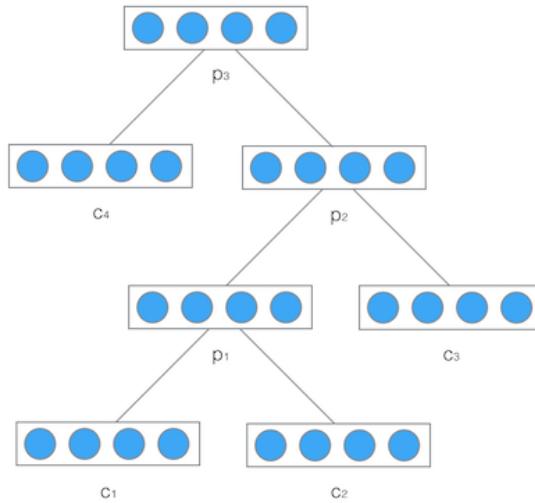


图 7.6: 神经元树

举个例子，我们使用递归神经网络将『两个外语学校的学生』映射为一个向量，如图??所示：

最后得到的向量  $p_3$  就是对整个句子『两个外语学校的学生』的表示。由于整个结构是递归的，不仅仅是根节点，事实上每个节点都是以其为根的子树的表示。比如，在左边的这棵树中，向量  $p_2$  是短语『外语学院的学生』的表示，而向量  $p_1$  是短语『外语学

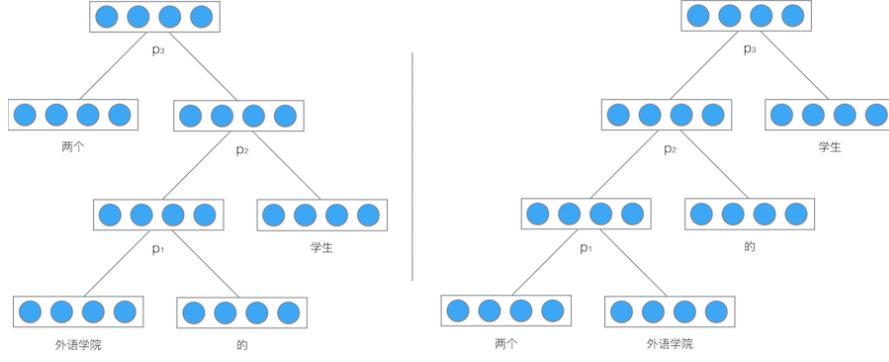


图 7.7: cross correlation

院的』的表示。

公式7.1就是递归神经网络的前向计算算法。它和全连接神经网络的计算没有什么区别，只是在输入的过程中需要根据输入的树结构依次输入每个子节点。

需要特别注意的是，递归神经网络的权重  $W$  和偏置项  $b$  在所有的节点都是共享的。

### 7.3 递归神经网络的训练

递归神经网络的训练算法和循环神经网络类似，两者不同之处在于，前者需要将残差  $\delta$  从根节点反向传播到各个子节点，而后者是将残差  $\delta$  从当前时刻  $t_k$  反向传播到初始时刻  $t_1$ 。

下面，我们介绍适用于递归神经网络的训练算法，也就是 **BPTS** 算法。

#### 7.3.1 误差项的传递

首先，我们先推导将误差从父节点传递到子节点的公式，如图7.8：

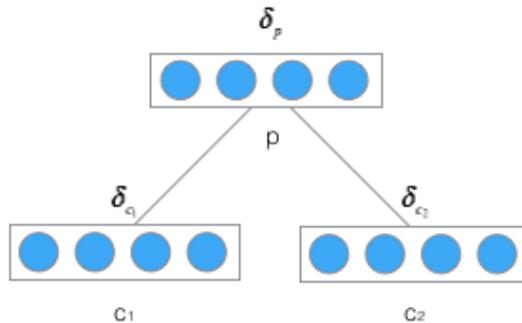


图 7.8: 神经元树

定义  $\delta_p$  为误差函数  $E$  相对于父节点  $p$  的加权输入  $net_p$  的导数，即：

$$\delta_p \stackrel{\text{def}}{=} \frac{\partial E}{\partial net_p}$$

设  $net_p$  是父节点的加权输入，则

$$net_p = W \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} + b$$

在上述式子里,  $net_p$ 、 $c_1$ 、 $c_2$  都是向量, 而  $W$  是矩阵。为了看清楚它们的关系, 我们将其展开:

$$\begin{bmatrix} net_{p1} \\ net_{p2} \\ \dots \\ net_{pn} \end{bmatrix} = \begin{bmatrix} w_{p_1c_{11}} & w_{p_1c_{12}} & \dots & w_{p_1c_{1n}} & w_{p_1c_{21}} & w_{p_1c_{22}} & \dots & w_{p_1c_{2n}} \\ w_{p_2c_{11}} & w_{p_2c_{12}} & \dots & w_{p_2c_{1n}} & w_{p_2c_{21}} & w_{p_2c_{22}} & \dots & w_{p_2c_{2n}} \\ \dots & \dots & & \dots & \dots & \dots & & \dots \\ w_{p_nc_{11}} & w_{p_nc_{12}} & \dots & w_{p_nc_{1n}} & w_{p_nc_{21}} & w_{p_nc_{22}} & \dots & w_{p_nc_{2n}} \end{bmatrix} \begin{bmatrix} c_{11} \\ c_{12} \\ \dots \\ c_{1n} \\ c_{21} \\ c_{22} \\ \dots \\ c_{2n} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix}$$

在上面的公式中,  $p_i$  表示父节点 p 的第  $i$  个分量;  $c_{1i}$  表示  $c_1$  子节点的第  $i$  个分量;  $c_{2i}$  表示  $c_2$  子节点的第  $i$  个分量;  $w_{p_ic_{jk}}$  表示子节点  $c_j$  的第  $k$  个分量到父节点 p 的第  $i$  个分量的权重。根据上面展开后的矩阵乘法形式, 我们不难看出, 对于子节点  $c_{jk}$  来说, 它会影响父节点所有的分量。因此, 我们求误差函数 E 对  $c_{jk}$  的导数时, 必须用到全导数公式, 也就是:

$$\frac{\partial E}{\partial c_{jk}} = \sum_i \frac{\partial E}{\partial net_{pi}} \frac{\partial net_{pi}}{\partial c_{jk}} = \sum_i \delta_{pi} w_{p_ic_{jk}}$$

有了上式, 我们就可以把它表示为矩阵形式, 从而得到一个向量化表达:

$$\frac{\partial E}{\partial c_j} = U_j \delta_p$$

其中, 矩阵  $U_j$  是从矩阵  $W$  中提取部分元素组成的矩阵。其单元为:

$$u_{jik} = w_{p_k c_{ji}}$$

上式看上去可能会让人晕菜, 从图7.9, 我们可以直观的看到  $U_j$  到底是啥。首先我们把  $W$  矩阵拆分为两个矩阵  $W_1$  和  $W_2$ 。

$$W = \begin{array}{c} W_1 \quad \quad \quad W_2 \\ \boxed{\begin{matrix} w_{p_1c_{11}} & w_{p_1c_{12}} & \dots & w_{p_1c_{1n}} \\ w_{p_2c_{11}} & w_{p_2c_{12}} & \dots & w_{p_2c_{1n}} \\ \dots & \dots & & \dots \\ w_{p_nc_{11}} & w_{p_nc_{12}} & \dots & w_{p_nc_{1n}} \end{matrix}} \quad \quad \quad \boxed{\begin{matrix} w_{p_1c_{21}} & w_{p_1c_{22}} & \dots & w_{p_1c_{2n}} \\ w_{p_2c_{21}} & w_{p_2c_{22}} & \dots & w_{p_2c_{2n}} \\ \dots & \dots & & \dots \\ w_{p_nc_{21}} & w_{p_nc_{22}} & \dots & w_{p_nc_{2n}} \end{matrix}} \end{array}$$

图 7.9: cross correlation

显然, 子矩阵  $W_1$  和  $W_2$  分别对应子节点  $c_1$  和  $c_2$  的到父节点 p 权重。则矩阵  $U_j$  为:

$$U_j = W_j^T$$

也就是说, 将误差项反向传递到相应子节点  $c_j$  的矩阵  $U_j$  就是其对应权重矩阵  $W_j$  的

转置。

现在，我们设  $net_{c_j}$  是子节点  $c_j$  的加权输入， $f$  是子节点  $c$  的激活函数，则：

$$c_j = f(net_{c_j})$$

这样，我们得到：

$$\delta_{c_j} = \frac{\partial E}{\partial net_{c_j}} = \frac{\partial E}{\partial c_j} \frac{\partial c_j}{\partial net_{c_j}} = W_j^T \delta_p \circ f'(net_{c_j})$$

如果我们将不同子节点  $c_j$  对应的误差项  $\delta_{c_j}$  连接成一个向量  $\delta_c = \begin{bmatrix} \delta_{c_1} \\ \delta_{c_2} \end{bmatrix}$ 。那么，上式可以写成：

$$\delta_c = W^T \delta_p \circ f'(net_c) \quad (7.2)$$

公式7.2就是将误差项从父节点传递到其子节点的公式。注意，上式中的  $net_c$  也是将两个子节点的加权输入  $net_{c_1}$  和  $net_{c_2}$  连在一起的向量。

有了传递一层的公式，我们就不难写出逐层传递的公式。

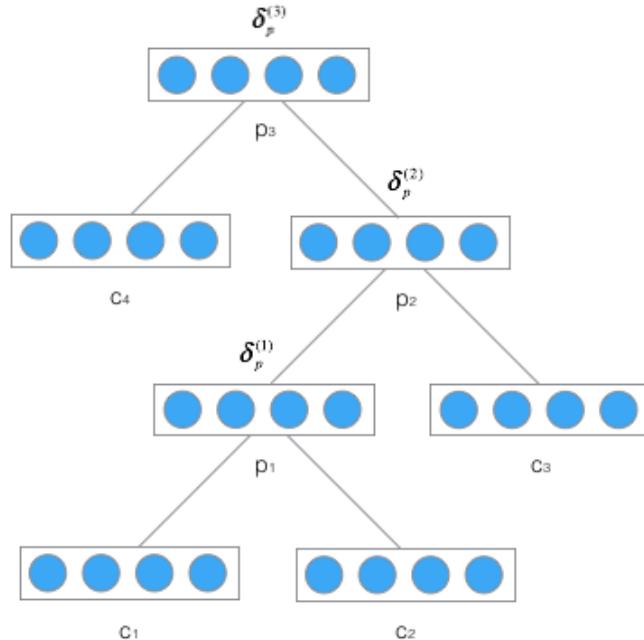


图 7.10: 神经元树

图7.10是在树型结构中反向传递误差项的全景图，反复应用公式7.2，在已知  $\delta_p^{(3)}$  的情况下，我们不难算出  $\delta_p^{(1)}$  为：

$$\delta^{(2)} = W^T \delta_p^{(3)} \circ f'(net^{(2)})$$

$$\delta_p^{(2)} = [\delta^{(2)}]_p$$

$$\delta^{(1)} = W^T \delta_p^{(2)} \circ f'(net^{(1)})$$

$$\delta_p^{(1)} = [\delta^{(1)}]_p$$

在上面的公式中， $\delta^{(2)} = \begin{bmatrix} \delta_c^{(2)} \\ \delta_p^{(2)} \end{bmatrix}$ ， $[\delta^{(2)}]_p$  表示取向量  $\delta^{(2)}$  属于节点  $p$  的部分。

### 7.3.2 权重梯度的计算

根据加权输入的计算公式：

$$\text{net}_p^{(l)} = Wc^{(l)} + b$$

其中， $\text{net}_p^{(l)}$  表示第 1 层的父节点的加权输入， $c^{(l)}$  表示第 1 层的子节点。 $W$  是权重矩阵， $b$  是偏置项。将其展开可得：

$$\text{net}_{p_j}^l = \sum_i w_{ji} c_i^{(l)} + b_j$$

那么，我们可以求得误差函数在第 1 层对权重的梯度为：

$$\frac{\partial E}{\partial w_{ji}^{(l)}} = \frac{\partial E}{\partial \text{net}_{p_j}^{(l)}} \frac{\partial \text{net}_{p_j}^{(l)}}{\partial w_{ji}^{(l)}} = \delta_{p_j}^{(l)} c_i^{(l)}$$

上式是针对一个权重项  $w_{ji}$  的公式，现在需要把它扩展为对所有的权重项的公式。我们可以把上式写成矩阵的形式（在下面的公式中， $m=2n$ ）：

$$\frac{\partial E}{\partial W^{(l)}} = \begin{bmatrix} \frac{\partial E}{\partial w_{11}^{(l)}} & \frac{\partial E}{\partial w_{12}^{(l)}} & \cdots & \frac{\partial E}{\partial w_{1m}^{(l)}} \\ \frac{\partial E}{\partial w_{21}^{(l)}} & \frac{\partial E}{\partial w_{22}^{(l)}} & \cdots & \frac{\partial E}{\partial w_{2m}^{(l)}} \\ \cdots & & & \\ \frac{\partial E}{\partial w_{n1}^{(l)}} & \frac{\partial E}{\partial w_{n2}^{(l)}} & \cdots & \frac{\partial E}{\partial w_{nm}^{(l)}} \end{bmatrix} = \begin{bmatrix} \delta_{p_1}^{(l)} c_1^{(l)} & \delta_{p_1}^{(l)} c_2^{(l)} & \cdots & \delta_{p_1}^{(l)} c_m^{(l)} \\ \delta_{p_2}^{(l)} c_1^{(l)} & \delta_{p_2}^{(l)} c_2^{(l)} & \cdots & \delta_{p_2}^{(l)} c_m^{(l)} \\ \cdots & & & \\ \delta_{p_n}^{(l)} c_1^{(l)} & \delta_{p_n}^{(l)} c_2^{(l)} & \cdots & \delta_{p_n}^{(l)} c_m^{(l)} \end{bmatrix} = \delta^{(l)} (c^{(l)})^T \quad (7.3)$$

公式 7.3 就是第 1 层权重项的梯度计算公式。我们知道，由于权重  $W$  是在所有层共享的，所以和循环神经网络一样，递归神经网络的最终的权重梯度是各个层权重梯度之和。即：

$$\frac{\partial E}{\partial W} = \sum_l \frac{\partial E}{\partial W^{(l)}} \quad (7.4)$$

因为循环神经网络的证明过程已经在第 4 章卷积神经网络一文中给出，因此，递归神经网络『为什么最终梯度是各层梯度之和』的证明就留给读者自行完成啦。

接下来，我们求偏置项  $b$  的梯度计算公式。先计算误差函数对第 1 层偏置项  $b^{(l)}$  的梯度：

$$\frac{\partial E}{\partial b_j^{(l)}} = \frac{\partial E}{\partial \text{net}_{p_j}^{(l)}} \frac{\partial \text{net}_{p_j}^{(l)}}{\partial b_j^{(l)}} = \delta_{p_j}^{(l)}$$

把上式扩展为矩阵的形式：

$$\frac{\partial E}{\partial b^{(l)}} = \begin{bmatrix} \frac{\partial E}{\partial b_1^{(l)}} \\ \frac{\partial E}{\partial b_2^{(l)}} \\ \cdots \\ \frac{\partial E}{\partial b_n^{(l)}} \end{bmatrix} = \begin{bmatrix} \delta_{p_1}^{(l)} \\ \delta_{p_2}^{(l)} \\ \cdots \\ \delta_{p_n}^{(l)} \end{bmatrix} = \delta_p^{(l)} \quad (7.5)$$

公式 7.5 是第 1 层偏置项的梯度，那么最终的偏置项梯度是各个层偏置项梯度之和，即：

$$\frac{\partial E}{\partial b} = \sum_l \frac{\partial E}{\partial b^{(l)}} \quad (7.6)$$

### 7.3.3 权重更新

如果使用梯度下降优化算法，那么权重更新公式为：

$$W \leftarrow W + \eta \frac{\partial E}{\partial W}$$

其中， $\eta$  是学习速率常数。把公式7.4带入到上式，即可完成权重的更新。同理，偏置项的更新公式为：

$$b \leftarrow b + \eta \frac{\partial E}{\partial b}$$

把公式7.6带入到上式，即可完成偏置项的更新。

这就是递归神经网络的训练算法 BPTS。由于我们有了前面几篇文章的基础，相信读者们理解 BPTS 算法也会比较容易。

## 7.4 编程实战：递归神经网络的实现

 **注意** 完整代码请参考 GitHub:[https://github.com/hanbt/learn\\_dl/blob/master/recursivne.py](https://github.com/hanbt/learn_dl/blob/master/recursivne.py)(python2.7)

现在，我们实现一个处理树型结构的递归神经网络。

在文件的开头，加入如下代码：

```
1 #!/usr/bin/env python
2 # -*- coding: UTF-8 -*-
3 import numpy as np
4 from cnn import IdentityActivator
```

上述四行代码非常简单，没有什么需要解释的。IdentityActivator 激活函数是在我们介绍卷积神经网络时写的，现在引用一下它。

我们首先定义一个树节点结构，这样，我们就可以用它保存卷积神经网络生成的整体树：

```
1 class TreeNode(object):
2     def __init__(self, data, children=[], children_data=[]):
3         self.parent = None
4         self.children = children
5         self.children_data = children_data
6         self.data = data
7         for child in children:
8             child.parent = self
```

接下来，我们把递归神经网络的实现代码都放在 RecursiveLayer 类中，下面是这个类的构造函数：

```
1 # 递归神经网络实现
2 class RecursiveLayer(object):
3     def __init__(self, node_width, child_count,
4                  activator, learning_rate):
5         ...
6
7         递归神经网络构造函数
8         node_width: 表示每个节点的向量的维度
9         child_count: 每个父节点有几个子节点
10        activator: 激活函数对象
11        learning_rate: 梯度下降算法学习率
12        ...
13
14        self.node_width = node_width
15        self.child_count = child_count
16        self.activator = activator
17        self.learning_rate = learning_rate
18
19        # 权重数组W
20        self.W = np.random.uniform(-1e-4, 1e-4,
21                                    (node_width, node_width * child_count))
22
23        # 偏置项b
24        self.b = np.zeros((node_width, 1))
25
26        # 递归神经网络生成的树的根节点
27        self.root = None
```

下面是前向计算的实现：

```
1     def forward(self, *children):
2         ...
3
4         前向计算
5         ...
6
7         children_data = self.concatenate(children)
8         parent_data = self.activator.forward(
9             np.dot(self.W, children_data) + self.b
10            )
11
12         self.root = TreeNode(parent_data, children
13                               , children_data)
```

forward 函数接收一系列的树节点对象作为输入，然后，递归神经网络将这些树节点作为子节点，并计算它们的父节点。最后，将计算的父节点保存在 self.root 变量中。

上面用到的 concatenate 函数，是将各个子节点中的数据拼接成一个长向量，其代码如下：

```
1     def concatenate(self, tree_nodes):
2         ...
3
4         将各个树节点中的数据拼接成一个长向量
5         ...
```

```
5     concat = np.zeros((0,1))
6     for node in tree_nodes:
7         concat = np.concatenate((concat, node.data))
8     return concat
```

下面是反向传播算法 BPTS 的实现：

```
1 def backward(self, parent_delta):
2     ...
3     BPTS 反向传播算法
4     ...
5     self.calc_delta(parent_delta, self.root)
6     self.W_grad, self.b_grad = self.calc_gradient(self.root)
7     def calc_delta(self, parent_delta, parent):
8         ...
9         计算每个节点的 delta
10        ...
11        parent.delta = parent_delta
12        if parent.children:
13            # 根据式2计算每个子节点的 delta
14            children_delta = np.dot(self.W.T, parent_delta) * (
15                self.activator.backward(parent.children_data)
16            )
17            # slices = [(子节点编号, 子节点 delta 起始位置, 子节点
18            #             delta 结束位置)]
19            slices = [(i, i * self.node_width,
20                      (i + 1) * self.node_width)
21                      for i in range(self.child_count)]
22            # 针对每个子节点, 递归调用 calc_delta 函数
23            for s in slices:
24                self.calc_delta(children_delta[s[1]:s[2]], parent.children[s[0]])
25        def calc_gradient(self, parent):
26            ...
27            计算每个节点权重的梯度, 并将它们求和, 得到最终的梯度
28            ...
29            W_grad = np.zeros((self.node_width,
30                               self.node_width * self.child_count))
31            b_grad = np.zeros((self.node_width, 1))
32            if not parent.children:
33                return W_grad, b_grad
34            parent.W_grad = np.dot(parent.delta, parent.children_data.
35            T)
35            parent.b_grad = parent.delta
```

```
36     W_grad += parent.W_grad
37     b_grad += parent.b_grad
38     for child in parent.children:
39         W, b = self.calc_gradient(child)
40         W_grad += W
41         b_grad += b
42     return W_grad, b_grad
```

在上述算法中，`calc_delta` 函数和 `calc_gradient` 函数分别计算各个节点的误差项  $\delta$  以及最终的梯度。它们都采用递归算法，先序遍历整个树，并逐一完成每个节点的计算。

下面是梯度下降算法的实现（没有 weight decay），这个非常简单：

```
1 def update(self):
2     ...
3     使用 SGD 算法 更新 权重
4     ...
5     self.W -= self.learning_rate * self.W_grad
6     self.b -= self.learning_rate * self.b_grad
```

以上就是递归神经网络的实现，总共 100 行左右，和上一篇文章的 LSTM 相比简单多了。

最后，我们用梯度检查来验证程序的正确性：

```
1 def gradient_check():
2     ...
3     梯度 检查
4     ...
5     # 设计一个误差函数，取所有节点输出项之和
6     error_function = lambda o: o.sum()
7     rnn = RecursiveLayer(2, 2, IdentityActivator(), 1e-3)
8     # 计算 forward 值
9     x, d = data_set()
10    rnn.forward(x[0], x[1])
11    rnn.forward(rnn.root, x[2])
12    # 求取 sensitivity map
13    sensitivity_array = np.ones((rnn.node_width, 1),
14                                dtype=np.float64)
15    # 计算梯度
16    rnn.backward(sensitivity_array)
17    # 检查梯度
18    epsilon = 10e-4
19    for i in range(rnn.W.shape[0]):
20        for j in range(rnn.W.shape[1]):
21            rnn.W[i, j] += epsilon
22            rnn.reset_state()
```

```

23         rnn.forward(x[0], x[1])
24         rnn.forward(rnn.root, x[2])
25         err1 = error_function(rnn.root.data)
26         rnn.W[i,j] -= 2*epsilon
27         rnn.reset_state()
28         rnn.forward(x[0], x[1])
29         rnn.forward(rnn.root, x[2])
30         err2 = error_function(rnn.root.data)
31         expect_grad = (err1 - err2) / (2 * epsilon)
32         rnn.W[i,j] += epsilon
33         print 'weights(%d,%d): expected - actual %.4e - %.4e' %
34             i, j, expect_grad, rnn.W_grad[i,j])
35     return rnn

```

下面是梯度检查的结果，完全正确，OH YEAH!

```

>>> rnn = recursive.gradient_check()
weights(0,0): expected - actual 3.8925e-05 - 3.8925e-05
weights(0,1): expected - actual -3.9918e-04 - -3.9918e-04
weights(0,2): expected - actual 4.9996e+00 - 4.9996e+00
weights(0,3): expected - actual 5.9995e+00 - 5.9995e+00
weights(1,0): expected - actual 8.0841e-05 - 8.0841e-05
weights(1,1): expected - actual -3.1535e-04 - -3.1535e-04
weights(1,2): expected - actual 4.9997e+00 - 4.9997e+00
weights(1,3): expected - actual 5.9996e+00 - 5.9996e+00

```

## 7.5 递归神经网络的应用

### 7.5.1 自然语言和自然场景解析

在自然语言处理任务中，如果我们能够实现一个解析器，将自然语言解析为语法树，那么毫无疑问，这将大大提升我们对自然语言的处理能力。解析器如图7.11所示：

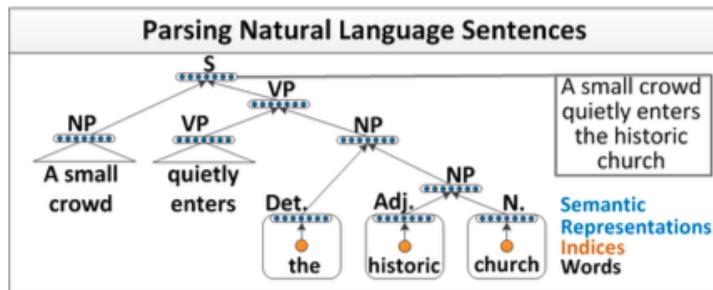


图 7.11：解析器

可以看出，递归神经网络能够完成句子的语法分析，并产生一个语法解析树。

除了自然语言之外，自然场景也具有可组合的性质。因此，我们可以用类似的模型完成自然场景的解析，如图7.12所示：

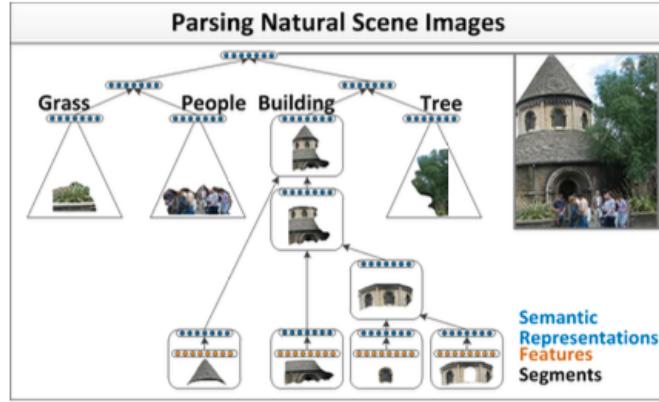


图 7.12: 自然场景的解析

两种不同的场景，可以用相同的递归神经网络模型来实现。我们以第一个场景，自然语言解析为例。

我们希望将一句话逐字输入到神经网络中，然后，神经网络返回一个解析好的树。为了做到这一点，我们需要给神经网络再加上一层，负责打分。分数越高，说明两个子节点结合更加紧密，分数越低，说明两个子节点结合更松散。如图7.13所示：

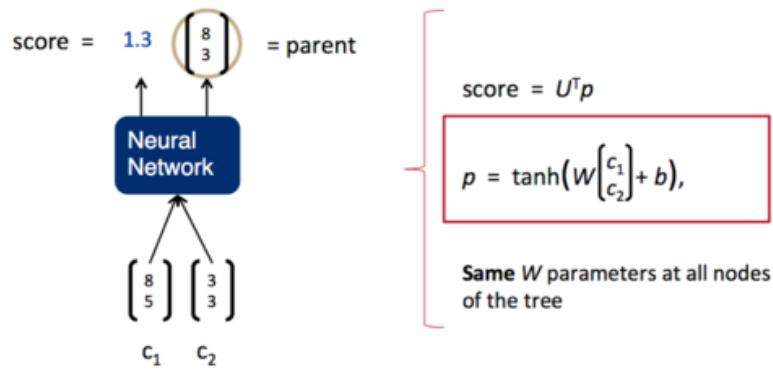


图 7.13: 两个子节点

一旦这个打分函数训练好了（也就是矩阵  $U$  的各项值变为合适的值），我们就可以利用贪心算法来实现句子的解析。第一步，我们先将词按照顺序两两输入神经网络，得到第一组打分：

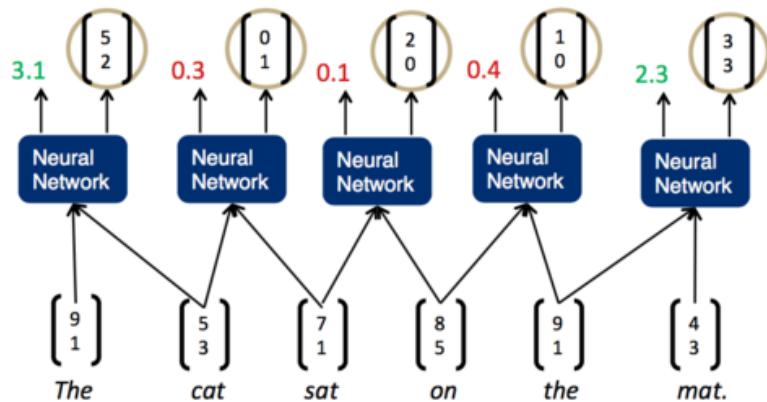


图 7.14: 输入神经网络

我们发现，现在分数最高的是第一组，The cat，说明它们的结合是最紧密的。这样，我们可以先将它们组合为一个节点。然后，再次两两计算相邻子节点的打分：

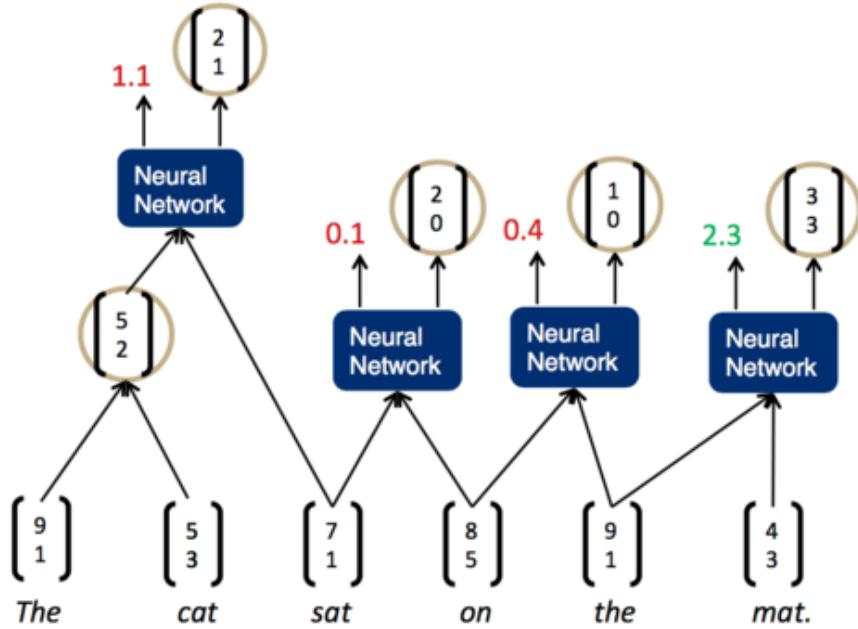


图 7.15：相邻子节点的打分

现在，分数最高的是最后一组，the mat。于是，我们将它们组合为一个节点，再两两计算相邻节点的打分：

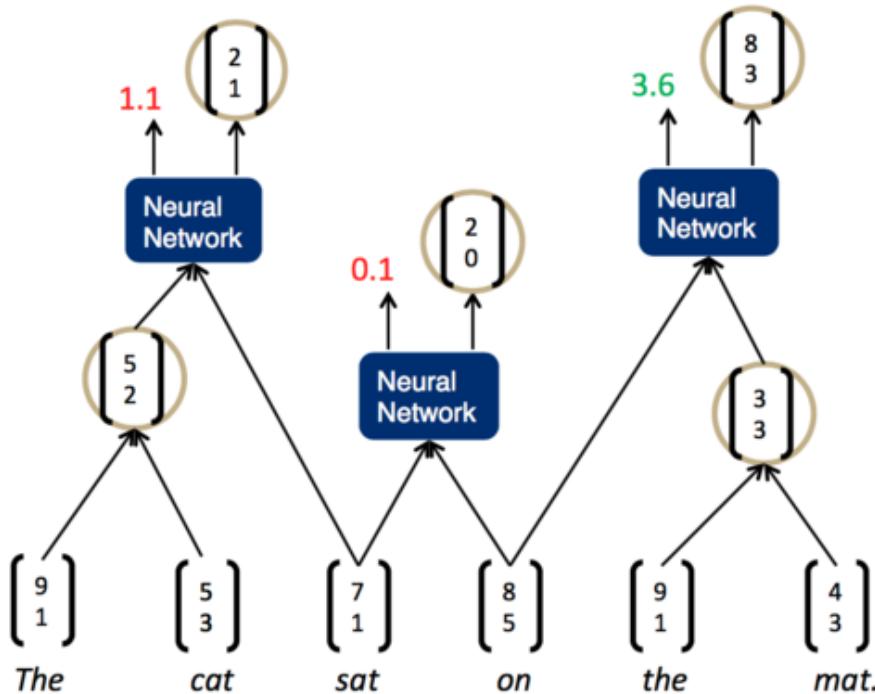


图 7.16：相邻子节点的打分

这时，我们发现最高的分数是 on the mat，把它们组合为一个节点，继续两两计算相邻节点的打分…… 最终，我们就能够得到整个解析树：

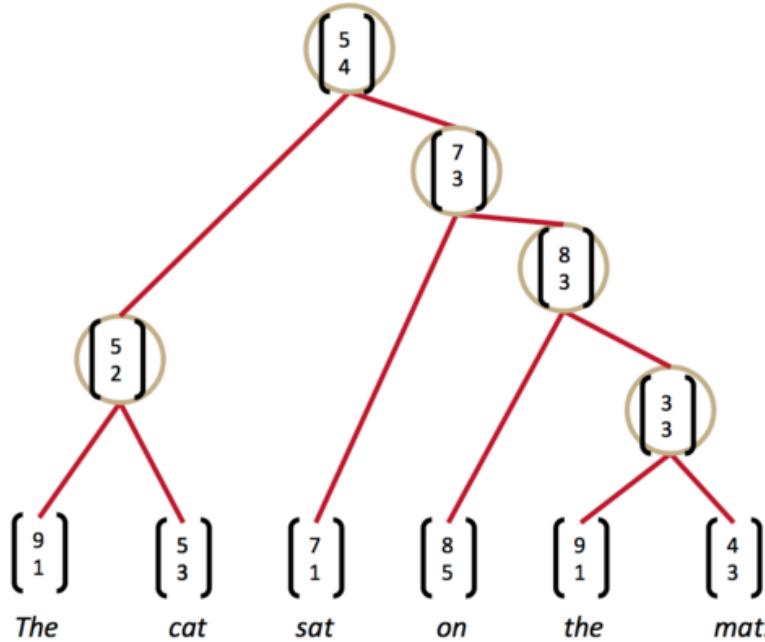


图 7.17: 整个解析树

现在，我们困惑这样牛逼的打分函数 score 是怎样训练出来的呢？我们需要定义一个目标函数。这里，我们使用 Max-Margin 目标函数。它的定义如下：

$$J(\theta) = \max(0, \sum_i \max_{y \in A(x_i)} (s(x_i, y) + \Delta(y, y_i)) - s(x_i, y_i))$$

在上式中， $x_i$ 、 $y_i$  分别表示第  $i$  个训练样本的输入和标签，注意这里的标签  $y_i$  是一棵解析树。 $s(x_i, y_i)$  就是打分函数  $s$  对第  $i$  个训练样本的打分。因为训练样本的标签肯定是正确的，我们希望  $s$  对它的打分越高越好，也就是  $s(x_i, y_i)$  越大越好。 $A(x_i)$  是所有可能的解析树的集合，而  $s(x_i, y)$  则是对某个可能的解析树  $y$  的打分。 $\Delta(y, y_i)$  是对错误的惩罚。也就是说，如果某个解析树  $y$  和标签  $y_i$  是一样的，那么  $\Delta(y, y_i)$  为 0，如果网络的输出错的越离谱，那么惩罚项  $\Delta(y, y_i)$  的值就越高。 $\max(s(x_i, y) + \Delta(y, y_i))$  表示所有树里面最高得分。在这里，惩罚项相当于 Margin，也就是我们虽然希望打分函数  $s$  对正确的树打分比对错误的树打分高，但也不要高过 Margin 的值。我们优化  $\theta$ ，使目标函数取最小值，即：

$$\theta = \arg \min_{\theta} J(\theta)$$

下面是惩罚函数  $\Delta$  的定义：

$$\Delta(y, y_i) = k \sum_{d \in N(y)} 1\{\text{subTree}(d) \notin y_i\}$$

上式中， $N(y)$  是树  $y$  节点的集合； $\text{subTree}(d)$  是以  $d$  为节点的子树。上式的含义是，如果以  $d$  为节点的子树没有出现在标签  $y_i$  中，那么函数值 +1。最终，惩罚函数的值，是树  $y$  中没有出现在树  $y_i$  中的子树的个数，再乘上一个系数  $k$ 。其实也就是关于两棵树差异的一个度量。

$s(x, y)$  是对一个样本最终的打分，它是对树  $y$  每个节点打分的总和。

$$s(x, y) = \sum_{n \in nodes(y)} s_n$$

具体细节，读者可以查阅『参考资料 3』的论文。

## 7.6 小结

我们在系列文章中已经介绍的全连接神经网络、卷积神经网络、循环神经网络和递归神经网络，在训练时都使用了**监督学习 (Supervised Learning)** 作为训练方法。在**监督学习**中，每个训练样本既包括输入特征  $x$ ，也包括标记  $y$ ，即样本  $d^{(i)} = \{x^{(i)}, y^{(i)}\}$ 。然而，很多情况下，我们无法获得形如  $\{x^{(i)}, y^{(i)}\}$  的样本，这时，我们就不能采用**监督学习**的方法。在接下来的几篇文章中，我们重点介绍另外一种学习方法：**增强学习 Reinforcement Learning**)。在了解**增强学习**的主要算法之后，我们还将介绍著名的围棋软件 **AlphaGo**，它是一个把**监督学习**和**增强学习**进行完美结合的案例。

