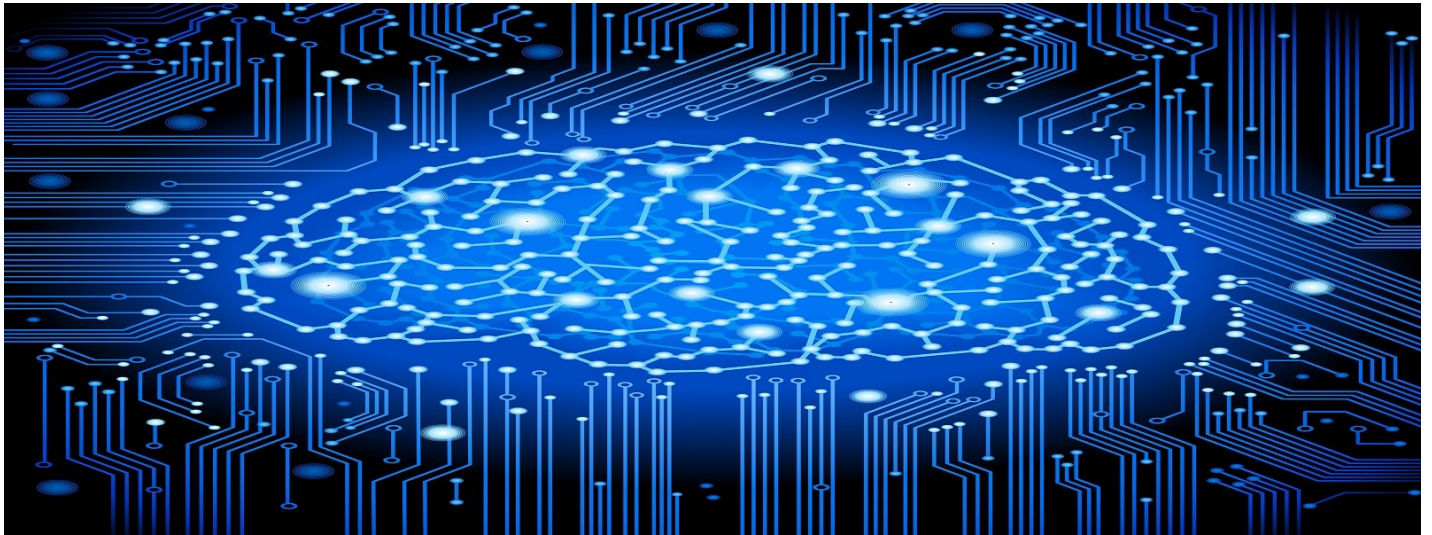


[\[关闭\]](#)

@hanbingtao 2017-08-28 19:54 字数 23656 阅读 198166

零基础入门深度学习(5) - 循环神经网络

机器学习 深度学习入门



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）。

文章列表

[零基础入门深度学习\(1\) - 感知器](#)[零基础入门深度学习\(2\) - 线性单元和梯度下降](#)[零基础入门深度学习\(3\) - 神经网络和反向传播算法](#)[零基础入门深度学习\(4\) - 卷积神经网络](#)[零基础入门深度学习\(5\) - 循环神经网络](#)[零基础入门深度学习\(6\) - 长短期记忆网络\(LSTM\)](#)[零基础入门深度学习\(7\) - 递归神经网络](#)

往期回顾

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**。

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。

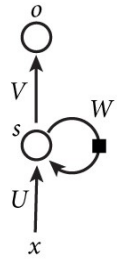
所以，该轮到RNN出场了，RNN理论上可以往前看(往后看)任意多个词。

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。

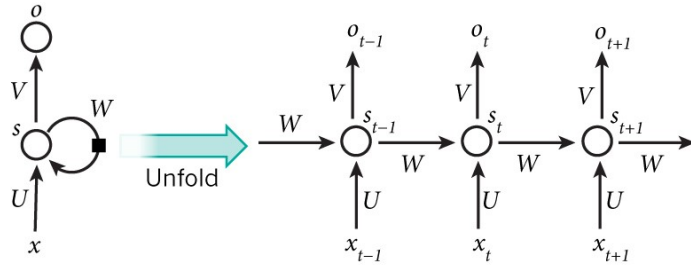
基本循环神经网络

下图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：



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（读者可以回到第三篇文章[零基础入门深度学习\(3\) - 神经网络和反向传播算法](#)，看看我们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下面这个样子：



现在看上去就比较清楚了，这个网络在时刻接收到输入 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：

$$\begin{aligned} o_t &= g(Vs_t) & (\neq 1) & (1) \\ s_t &= f(Ux_t + Ws_{t-1}) & (\neq 2) & (2) \end{aligned}$$

式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。式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。

如果反复把式2带入到式1，我们将得到：

$$\begin{aligned} o_t &= g(Vs_t) & (3) \\ &= Vf(Ux_t + Ws_{t-1}) & (4) \\ &= Vf(Ux_t + Wf(Ux_{t-1} + Ws_{t-2})) & (5) \\ &= Vf(Ux_t + Wf(Ux_{t-1} + Wf(Ux_{t-2} + Ws_{t-3}))) & (6) \\ &= Vf(Ux_t + Wf(Ux_{t-1} + Wf(Ux_{t-2} + Wf(Ux_{t-3} + \dots)))) & (7) \end{aligned}$$

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 $x_t, x_{t-1}, x_{t-2}, x_{t-3}, \dots$ 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。

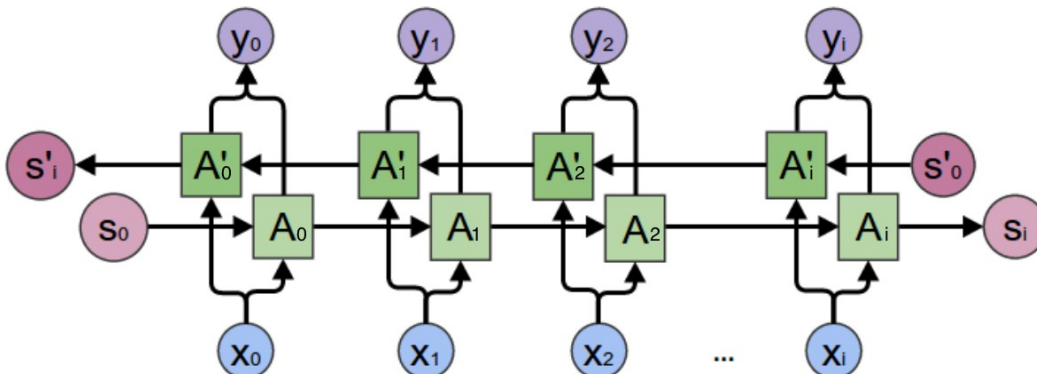
双向循环神经网络

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，如下图所示：



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。我们先考虑上图中， y_2 的计算。

从上图可以看出，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：

$$\begin{aligned} A_2 &= f(WA_1 + Ux_2) & (8) \\ A'_2 &= f(W'A'_3 + U'x_2) & (9) \end{aligned}$$

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 a_t 与 a_{t-1} 有关；反向计算时，隐藏层的值 a'_t 与 a'_{t+1} 有关；最终的输出取决于正向和反向计算的加和。现在，我们仿照式1和式2，写出双向循环神经网络的计算方法：

$$\alpha_t = g(Vs_t + V'a'_t) \quad (10)$$

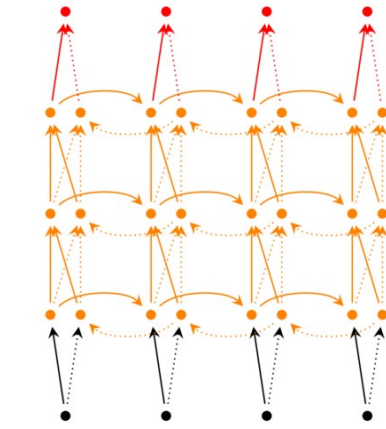
$$s_t = f(Ux_t + Ws_{t-1}) \quad (11)$$

$$a'_t = f(U'x_t + W'a'_{t+1}) \quad (12)$$

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**。

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**。如下图所示：



我们把第*i*个隐藏层的值表示为 $s_i^{(i)}$ 、 $a'_i{}^{(i)}$ ，则**深度循环神经网络**的计算方式可以表示为：

$$\alpha_t = g(V^{(i)}s_t^{(i)} + V'^{(i)}a'_t{}^{(i)}) \quad (13)$$

$$s_t^{(i)} = f(U^{(i)}s_t^{(i-1)} + W^{(i)}s_{t-1}) \quad (14)$$

$$a'_t{}^{(i)} = f(U'^{(i)}a'_t{}^{(i-1)} + W'^{(i)}a'_{t+1}) \quad (15)$$

$$\dots \quad (16)$$

$$s_t^{(1)} = f(U^{(1)}x_t + W^{(1)}s_{t-1}) \quad (17)$$

$$a'_t{}^{(1)} = f(U'^{(1)}x_t + W'^{(1)}a'_{t+1}) \quad (18)$$

循环神经网络的训练

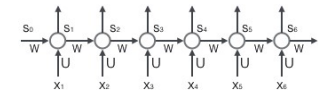
循环神经网络的训练算法：BPTT

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** δ_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。

循环层如下图所示：



前向计算

使用前面的**式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* × *m*，矩阵W的维度是*n* × *n*。下面是上式展开成矩阵的样子，看起来更直观一些：

$$\begin{bmatrix} s_1^t \\ s_2^t \\ \vdots \\ s_n^t \end{bmatrix} = f\left(\begin{bmatrix} u_{11} & u_{12} & \dots & u_{1m} \\ u_{21} & u_{22} & \dots & u_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ u_{n1} & u_{n2} & \dots & u_{nm} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} + \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nn} \end{bmatrix} \begin{bmatrix} s_1^{t-1} \\ s_2^{t-1} \\ \vdots \\ s_n^{t-1} \end{bmatrix} \right) \quad (19)$$

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*i*时刻的第*i*个神经元到**循环层**第*i*时刻的第*j*个神经元的权重。

误差项的计算

BTPP算法将第*i*层*t*时刻的**误差项** δ_i^t 沿两个方向传播，一个方向是其传递到上一层网络，得到 δ_i^{t-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 δ_i^1 ，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：

$$net_t = Ux_t + Ws_{t-1} \quad (20)$$

$$s_{t-1} = f(net_{t-1}) \quad (21)$$

因此：

$$\frac{\partial net_t}{\partial net_{t-1}} = \frac{\partial net_t}{\partial s_{t-1}} \frac{\partial s_{t-1}}{\partial net_{t-1}} \quad (22)$$

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：

$$\frac{\partial net_t}{\partial s_{t-1}} = \begin{bmatrix} \frac{\partial net_1^t}{\partial s_1^{t-1}} & \frac{\partial net_1^t}{\partial s_2^{t-1}} & \cdots & \frac{\partial net_1^t}{\partial s_n^{t-1}} \\ \frac{\partial net_2^t}{\partial s_1^{t-1}} & \frac{\partial net_2^t}{\partial s_2^{t-1}} & \cdots & \frac{\partial net_2^t}{\partial s_n^{t-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial net_n^t}{\partial s_1^{t-1}} & \frac{\partial net_n^t}{\partial s_2^{t-1}} & \cdots & \frac{\partial net_n^t}{\partial s_n^{t-1}} \end{bmatrix} \quad (23)$$

$$= \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nn} \end{bmatrix} \quad (24)$$

$$= W \quad (25)$$

同理，上式第二项也是一个Jacobian矩阵：

$$\frac{\partial s_{t-1}}{\partial net_{t-1}} = \begin{bmatrix} \frac{\partial s_1^{t-1}}{\partial net_1^{t-1}} & \frac{\partial s_1^{t-1}}{\partial net_2^{t-1}} & \cdots & \frac{\partial s_1^{t-1}}{\partial net_n^{t-1}} \\ \frac{\partial s_2^{t-1}}{\partial net_1^{t-1}} & \frac{\partial s_2^{t-1}}{\partial net_2^{t-1}} & \cdots & \frac{\partial s_2^{t-1}}{\partial net_n^{t-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial s_n^{t-1}}{\partial net_1^{t-1}} & \frac{\partial s_n^{t-1}}{\partial net_2^{t-1}} & \cdots & \frac{\partial s_n^{t-1}}{\partial net_n^{t-1}} \end{bmatrix} \quad (26)$$

$$= \begin{bmatrix} f'(net_1^{t-1}) & 0 & \cdots & 0 \\ 0 & f'(net_2^{t-1}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'(net_n^{t-1}) \end{bmatrix} \quad (27)$$

$$= \text{diag}[f'(net_{t-1})] \quad (28)$$

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

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

最后，将两项合在一起，可得：

$$\frac{\partial net_t}{\partial net_{t-1}} = \frac{\partial net_t}{\partial s_{t-1}} \frac{\partial s_{t-1}}{\partial net_{t-1}} \quad (29)$$

$$= W \text{diag}[f'(net_{t-1})] \quad (30)$$

$$= \begin{bmatrix} w_{11}f'(net_1^{t-1}) & w_{12}f'(net_2^{t-1}) & \cdots & w_{1n}f'(net_n^{t-1}) \\ w_{21}f'(net_1^{t-1}) & w_{22}f'(net_2^{t-1}) & \cdots & w_{2n}f'(net_n^{t-1}) \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1}f'(net_1^{t-1}) & w_{n2}f'(net_2^{t-1}) & \cdots & w_{nn}f'(net_n^{t-1}) \end{bmatrix} \quad (31)$$

上式描述了将 δ 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 δ_k ：

$$\delta_k^T = \frac{\partial E}{\partial net_k} \quad (32)$$

$$= \frac{\partial E}{\partial net_t} \frac{\partial net_t}{\partial net_k} \quad (33)$$

$$= \frac{\partial E}{\partial net_t} \frac{\partial net_t}{\partial net_{t-1}} \frac{\partial net_{t-1}}{\partial net_{t-2}} \cdots \frac{\partial net_{t-1}}{\partial net_k} \quad (34)$$

$$= W \text{diag}[f'(net_{t-1})] W \text{diag}[f'(net_{t-2})] \cdots W \text{diag}[f'(net_k)] \delta_t^T \quad (35)$$

$$= \delta_t^T \prod_{i=k}^{t-1} W \text{diag}[f'(net_i)] \quad (\text{式3}) \quad (36)$$

式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，这在前面的文章[零基础入门深度学习\(3\) - 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^t 与上一层的加权输入 net^{t-1} 关系如下：

$$net_t^i = Ua_{t-1}^{i-1} + Ws_{t-1}^i \quad (37)$$

$$a_{t-1}^{i-1} = f^{i-1}(net_{t-1}^{i-1}) \quad (38)$$

上式中 net_t^i 是第 i 层神经元的加权输入(假设第 i 层是循环层)； net_{t-1}^{i-1} 是第 $i-1$ 层神经元的加权输入； a_{t-1}^{i-1} 是第 $i-1$ 层神经元的输出； f^{i-1} 是第 $i-1$ 层的激活函数。

$$\frac{\partial net_t^i}{\partial net_{t-1}^{i-1}} = \frac{\partial net_t^i}{\partial a_{t-1}^{i-1}} \frac{\partial a_{t-1}^{i-1}}{\partial net_{t-1}^{i-1}} \quad (39)$$

$$= U \text{diag}[f^{i-1}(net_{t-1}^{i-1})] \quad (40)$$

所以，

$$(\delta_t^{i-1})^T = \frac{\partial E}{\partial net_{t-1}^{i-1}} \quad (41)$$

$$= \frac{\partial E}{\partial net_t^i} \frac{\partial net_t^i}{\partial net_{t-1}^{i-1}} \quad (42)$$

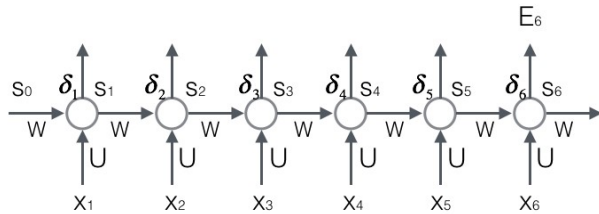
$$= (\delta_t^i)^T U \text{diag}[f^{i-1}(net_{t-1}^{i-1})] \quad (\text{式4}) \quad (43)$$

式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 $\frac{\partial E}{\partial W}$ 。



上图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：循环层的输出值 s_t ，以及误差项 δ_t 。

回忆一下我们在文章[零基础入门深度学习\(3\) - 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 δ_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 $\nabla_{W_t} E$ ：

$$\nabla_{W_t} E = \begin{bmatrix} \delta_t^1 s_1^{t-1} & \delta_t^1 s_2^{t-1} & \dots & \delta_t^1 s_n^{t-1} \\ \delta_t^2 s_1^{t-1} & \delta_t^2 s_2^{t-1} & \dots & \delta_t^2 s_n^{t-1} \\ \vdots & \vdots & \ddots & \vdots \\ \delta_t^n s_1^{t-1} & \delta_t^n s_2^{t-1} & \dots & \delta_t^n s_n^{t-1} \end{bmatrix} \quad (\text{式5})$$

在式5中， δ_t^i 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式5。

我们知道：

$$net_t = Ux_t + Ws_{t-1} \quad (44)$$

$$\begin{bmatrix} net_1^t \\ net_2^t \\ \vdots \\ net_n^t \end{bmatrix} = Ux_t + \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nn} \end{bmatrix} \begin{bmatrix} s_1^{t-1} \\ s_2^{t-1} \\ \vdots \\ s_n^{t-1} \end{bmatrix} \quad (45)$$

$$= Ux_t + \begin{bmatrix} w_{11}s_1^{t-1} + w_{12}s_2^{t-1} + \dots + w_{1n}s_n^{t-1} \\ w_{21}s_1^{t-1} + w_{22}s_2^{t-1} + \dots + w_{2n}s_n^{t-1} \\ \vdots \\ w_{n1}s_1^{t-1} + w_{n2}s_2^{t-1} + \dots + w_{nn}s_n^{t-1} \end{bmatrix} \quad (46)$$

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 w_{μ} 求导。通过观察上式我们可以看到 w_{μ} 只与 net_j^t 有关，所以：

$$\frac{\partial E}{\partial w_{\mu}} = \frac{\partial E}{\partial net_j^t} \frac{\partial net_j^t}{\partial w_{\mu}} \quad (47)$$

$$= \delta_j^t s_i^{t-1} \quad (48)$$

按照上面的规律就可以生成式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 $\nabla_{W_t} E$ ，最终的梯度 $\nabla_W E$ 是各个时刻的梯度之和：

$$\nabla_W E = \sum_{t=1}^T \nabla_{W_t} E \quad (49)$$

$$= \begin{bmatrix} \delta_1^1 s_1^0 & \delta_1^1 s_2^0 & \dots & \delta_1^1 s_n^0 \\ \delta_2^1 s_1^0 & \delta_2^1 s_2^0 & \dots & \delta_2^1 s_n^0 \\ \vdots & \vdots & \ddots & \vdots \\ \delta_n^1 s_1^0 & \delta_n^1 s_2^0 & \dots & \delta_n^1 s_n^0 \end{bmatrix} + \dots + \begin{bmatrix} \delta_1^T s_1^{T-1} & \delta_1^T s_2^{T-1} & \dots & \delta_1^T s_n^{T-1} \\ \delta_2^T s_1^{T-1} & \delta_2^T s_2^{T-1} & \dots & \delta_2^T s_n^{T-1} \\ \vdots & \vdots & \ddots & \vdots \\ \delta_n^T s_1^{T-1} & \delta_n^T s_2^{T-1} & \dots & \delta_n^T s_n^{T-1} \end{bmatrix} \quad (\text{式6}) \quad (50)$$

式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。感兴趣的同学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：

$$\frac{\partial net_t}{\partial W} = \frac{\partial W}{\partial W} f(net_{t-1}) + W \frac{\partial f(net_{t-1})}{\partial W}$$

我们最终需要计算的是 $\nabla_W E$ ：

$$\nabla_W E = \frac{\partial E}{\partial W} \quad (51)$$

$$= \frac{\partial E}{\partial net_t} \frac{\partial net_t}{\partial W} \quad (52)$$

$$= \delta_t^t \frac{\partial W}{\partial W} f(net_{t-1}) + \delta_t^t W \frac{\partial f(net_{t-1})}{\partial W} \quad (\text{式7}) \quad (53)$$

我们先计算式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：

$$\frac{\partial W}{\partial W} = \begin{bmatrix} \frac{\partial w_{11}}{\partial W} & \frac{\partial w_{12}}{\partial W} & \cdots & \frac{\partial w_{1n}}{\partial W} \\ \frac{\partial w_{21}}{\partial W} & \frac{\partial w_{22}}{\partial W} & \cdots & \frac{\partial w_{2n}}{\partial W} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial w_{n1}}{\partial W} & \frac{\partial w_{n2}}{\partial W} & \cdots & \frac{\partial w_{nn}}{\partial W} \end{bmatrix} \quad (54)$$

$$= \begin{bmatrix} \begin{bmatrix} \frac{\partial w_{11}}{\partial W} & \frac{\partial w_{12}}{\partial W} & \cdots & \frac{\partial w_{1n}}{\partial W} \\ \frac{\partial w_{21}}{\partial W} & \frac{\partial w_{22}}{\partial W} & \cdots & \frac{\partial w_{2n}}{\partial W} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial w_{n1}}{\partial W} & \frac{\partial w_{n2}}{\partial W} & \cdots & \frac{\partial w_{nn}}{\partial W} \end{bmatrix} & \begin{bmatrix} \frac{\partial w_{12}}{\partial W} & \frac{\partial w_{13}}{\partial W} & \cdots & \frac{\partial w_{1n}}{\partial W} \\ \frac{\partial w_{22}}{\partial W} & \frac{\partial w_{23}}{\partial W} & \cdots & \frac{\partial w_{2n}}{\partial W} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial w_{n2}}{\partial W} & \frac{\partial w_{n3}}{\partial W} & \cdots & \frac{\partial w_{nn}}{\partial W} \end{bmatrix} & \cdots \end{bmatrix} \quad (55)$$

$$= \begin{bmatrix} \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} & \begin{bmatrix} 0 & 1 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} & \cdots \end{bmatrix} \quad (56)$$

接下来, 我们知道 $a_{t-1} = f(\text{net}_{t-1})$, 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 δ_t^T , 最终得到一个矩阵:

$$\delta_t^T \frac{\partial W}{\partial W} f(\text{net}_{t-1}) = \delta_t^T \frac{\partial W}{\partial W} a_{t-1} \quad (57)$$

$$= \delta_t^T \begin{bmatrix} \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} & \begin{bmatrix} 0 & 1 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} & \cdots \end{bmatrix} \begin{bmatrix} a_{t-1}^1 \\ a_{t-1}^2 \\ \vdots \\ a_{t-1}^n \end{bmatrix} \quad (58)$$

$$= \delta_t^T \begin{bmatrix} \begin{bmatrix} a_{t-1}^1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} & \begin{bmatrix} a_{t-1}^2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} & \cdots \end{bmatrix} \quad (59)$$

$$= \begin{bmatrix} \delta_1^t & \delta_2^t & \cdots & \delta_n^t \end{bmatrix} \begin{bmatrix} \begin{bmatrix} a_{t-1}^1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} & \begin{bmatrix} a_{t-1}^2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} & \cdots \end{bmatrix} \quad (60)$$

$$= \begin{bmatrix} \delta_1^t a_{t-1}^1 & \delta_1^t a_{t-1}^2 & \cdots & \delta_1^t a_{t-1}^n \\ \delta_2^t a_{t-1}^1 & \delta_2^t a_{t-1}^2 & \cdots & \delta_2^t a_{t-1}^n \\ \vdots & \vdots & \ddots & \vdots \\ \delta_n^t a_{t-1}^1 & \delta_n^t a_{t-1}^2 & \cdots & \delta_n^t a_{t-1}^n \end{bmatrix} \quad (61)$$

$$= \nabla_{W_t} E \quad (62)$$

接下来, 我们计算式7加号右边的部分:

$$\delta_t^T W \frac{\partial f(\text{net}_{t-1})}{\partial W} = \delta_t^T W \frac{\partial f(\text{net}_{t-1})}{\partial \text{net}_{t-1}} \frac{\partial \text{net}_{t-1}}{\partial W} \quad (63)$$

$$= \delta_t^T W f'(\text{net}_{t-1}) \frac{\partial \text{net}_{t-1}}{\partial W} \quad (64)$$

$$= \delta_t^T \frac{\partial \text{net}_t}{\partial \text{net}_{t-1}} \frac{\partial \text{net}_{t-1}}{\partial W} \quad (65)$$

$$= \delta_{t-1}^T \frac{\partial \text{net}_{t-1}}{\partial W} \quad (66)$$

于是, 我们得到了如下递推公式:

$$\nabla_W E = \frac{\partial E}{\partial W} \quad (67)$$

$$= \frac{\partial E}{\partial \text{net}_t} \frac{\partial \text{net}_t}{\partial W} \quad (68)$$

$$= \nabla_{W_t} E + \delta_{t-1}^T \frac{\partial \text{net}_{t-1}}{\partial W} \quad (69)$$

$$= \nabla_{W_t} E + \nabla_{W_{t-1}} E + \delta_{t-2}^T \frac{\partial \text{net}_{t-2}}{\partial W} \quad (70)$$

$$= \nabla_{W_t} E + \nabla_{W_{t-1}} E + \cdots + \nabla_{W_1} E \quad (71)$$

$$= \sum_{k=1}^t \nabla_{W_k} E \quad (72)$$

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。

$$\nabla_U E = \begin{bmatrix} \delta_1^t x_1^t & \delta_1^t x_2^t & \cdots & \delta_1^t x_n^t \\ \delta_2^t x_1^t & \delta_2^t x_2^t & \cdots & \delta_2^t x_n^t \\ \vdots & \vdots & \ddots & \vdots \\ \delta_n^t x_1^t & \delta_n^t x_2^t & \cdots & \delta_n^t x_n^t \end{bmatrix} \quad (式8)$$

式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:

$$\nabla_U E = \sum_{k=1}^t \nabla_{U_k} E$$

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。

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。我们根据式3可得:

$$\delta_t^r = \delta_t^l \prod_{i=k}^{t-1} W \text{diag}[f'(net_i)] \quad (73)$$

$$\|\delta_t^r\| \leq \|\delta_t^l\| \prod_{i=k}^{t-1} \|W\| \|\text{diag}[f'(net_i)]\| \quad (74)$$

$$\leq \|\delta_t^l\| (\beta_W \beta_f)^{t-k} \quad (75)$$

上式的 β 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 β 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：

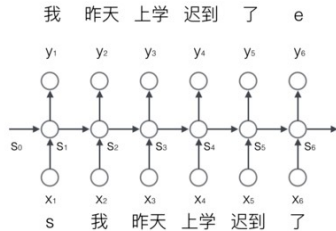
1. 合理的初始化权重值。初始权重值，使每个神经元尽可能不要取极大或极小值，以躲开梯度消失的区域。
2. 使用relu代替sigmoid和tanh作为激活函数。原理请参考上一篇文章[零基础入门深度学习\(4\) - 卷积神经网络的激活函数](#)一节。
3. 使用其他结构的RNNs，比如长短期记忆网络（LSTM）和Gated Recurrent Unit（GRU），这是最流行的做法。我们将在以后的文章中介绍这两种网络。

RNN的应用举例——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

神经网络的输出如下图所示：



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。

向量化

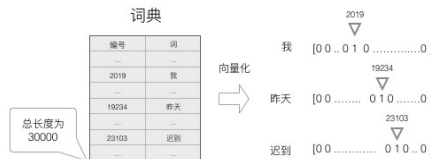
我们知道，神经网络的输入和输出都是向量，为了让语言模型能够被神经网络处理，我们必须把词表达为向量的形式，这样神经网络才能处理它。

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：

1. 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。
2. 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：

$$v_j = \begin{cases} 1 & j = i \\ 0 & j \neq i \end{cases} \quad (76)$$

上面这个公式的含义，可以用下面的图来直观地表示：



使用这种向量化方法，我们就得到了一个高维、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。如下图所示：



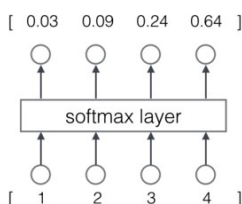
Softmax层

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层如下图所示：



从上图我们可以看到，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：

$$y_1 = \frac{e^{0.1}}{\sum_k e^{o_k}} \quad (77)$$

$$= \frac{e^1}{e^1 + e^2 + e^3 + e^4} \quad (78)$$

$$= 0.03 \quad (79)$$

$$y_2 = \frac{e^2}{e^1 + e^2 + e^3 + e^4} \quad (80)$$

$$= 0.09 \quad (81)$$

$$y_3 = \frac{e^3}{e^1 + e^2 + e^3 + e^4} \quad (82)$$

$$= 0.24 \quad (83)$$

$$y_4 = \frac{e^4}{e^1 + e^2 + e^3 + e^4} \quad (84)$$

$$= 0.64 \quad (85)$$

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。

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：

输入 标签

s 我

我 昨天

昨天 上学

上学 迟到

迟到了

了 c

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。

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E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）：

$$L = -\frac{1}{N} \sum_{n \in N} y_n \log o_n \quad (86)$$

$$= -y_1 \log o_1 \quad (87)$$

$$= -(1 * \log 0.03 + 0 * \log 0.09 + 0 * \log 0.24 + 0 * \log 0.64) \quad (88)$$

$$= 3.51 \quad (89)$$

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[参考文献7](#)。

RNN的实现

完整代码请参考GitHub: 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。我们复用了上一篇文章[零基础入门深度学习\(4\) - 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.
5.         self.input_width = input_width
6.         self.state_width = state_width
7.         self.activator = activator
8.         self.learning_rate = learning_rate
9.         self.times = 0 # 当前时刻初始化为0
10.        self.state_list = [] # 保存各个时刻的state
11.        self.state_list.append(np.zeros(
12.            (state_width, 1))) # 初始化s0
13.        self.U = np.random.uniform(-1e-4, 1e-4,
14.            (state_width, input_width)) # 初始化U
15.        self.W = np.random.uniform(-1e-4, 1e-4,
16.            (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.
9.     def calc_delta(self, sensitivity_array, activator):
10.        self.delta_list = [] # 用来保存各个时刻的误差项
11.        for i in range(self.times):
12.            self.delta_list.append(np.zeros(
13.                (self.state_width, 1)))
14.            self.delta_list.append(sensitivity_array)
15.            # 迭代计算每个时刻的误差项
16.            for k in range(self.times - 1, 0, -1):
17.                self.calc_delta_k(k, activator)
18.
19.    def calc_delta_k(self, k, activator):
20.        '''
21.        根据k+1时刻的delta计算k时刻的delta
22.        '''
23.        state = self.state_list[k+1].copy()
24.        element_wise_op(self.state_list[k+1],
25.                        activator.backward)
26.        self.delta_list[k] = np.dot(
27.            np.dot(self.delta_list[k+1].T, self.W),
28.            np.diag(state[:,0])).T
29.
30.    def calc_gradient(self):
31.        self.gradient_list = [] # 保存各个时刻的权重梯度
32.        for t in range(self.times + 1):
33.            self.gradient_list.append(np.zeros(
34.                (self.state_width, self.state_width)))
35.            for t in range(self.times, 0, -1):
36.                self.calc_gradient_t(t)
37.            # 实际的梯度是各个时刻梯度之和
38.            self.gradient = reduce(
39.                lambda a, b: a + b, self.gradient_list,
40.                self.gradient_list[0]) # {0}被初始化为0且没有被修改过
41.
42.    def calc_gradient_t(self, t):
43.        '''
44.        计算每个时刻t权重的梯度
45.        '''
46.        gradient = np.dot(self.delta_list[t],
47.                          self.state_list[t-1].T)
48.        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W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0
3.         self.state_list = [] # 保存各个时刻的state
4.         self.state_list.append(np.zeros(
5.             (self.state_width, 1))) # 初始化为0
```

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.
8.         rl = RecurrentLayer(3, 2, IdentityActivator(), 1e-3)
9.
10.        # 计算forward值
11.        x, d = data.set()
12.        rl.forward(x[0])
13.        rl.forward(x[1])
14.
15.        # 求取sensitivity map
16.        sensitivity_array = np.ones(rl.state_list[-1].shape,
17.                                    dtype=np.float64)
18.
19.        # 计算梯度
20.        rl.backward(sensitivity_array, IdentityActivator())
21.
22.        # 检查梯度
23.        epsilon = 10e-4
24.        for i in range(rl.W.shape[0]):
25.            for j in range(rl.W.shape[1]):
26.                rl.W[i,j] += epsilon
27.                rl.reset_state()
28.                rl.forward(x[0])
29.                rl.forward(x[1])
30.                err1 = error_function(rl.state_list[-1])
31.                rl.W[i,j] -= 2*epsilon
32.                rl.reset_state()
33.                rl.forward(x[0])
34.                rl.forward(x[1])
35.                err2 = error_function(rl.state_list[-1])
36.                expect_grad = (err1 - err2) / (2 * epsilon)
37.                print 'weights(%d,%d): expected - actual %f - %f' % (
38.                    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 - actual 0.000013 - 0.000013
weights(0,1): expected - actual 0.000383 - 0.000383
weights(1,0): expected - actual 0.000013 - 0.000013
weights(1,1): expected - actual 0.000383 - 0.000383
```

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：长短期记忆网络。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期记忆网络吧。



参考资料

1. [RECURRENT NEURAL NETWORKS TUTORIAL](#)
2. [Understanding LSTM Networks](#)
3. [The Unreasonable Effectiveness of Recurrent Neural Networks](#)
4. [Attention and Augmented Recurrent Neural Networks](#)
5. [On the difficulty of training recurrent neural networks](#), Bengio et al.
6. [Recurrent neural network based language model](#), Mikolov et al.
7. [Neural Network Classification, Categorical Data, Softmax Activation, and Cross Entropy Error](#), McCaffrey

•

内容目录

◦ 零基础入门深度学习(5) - 循环神经网络

- [文章列表](#)
- [往期回顾](#)
- [语言模型](#)
- [循环神经网络是啥](#)
 - [基本循环神经网络](#)
 - [双向循环神经网络](#)
 - [深度循环神经网络](#)
- [循环神经网络的训练](#)
 - [循环神经网络的训练算法: BPTT](#)
 - [前向计算](#)
 - [误差项的计算](#)
 - [权重梯度的计算](#)
 - [RNN的梯度爆炸和消失问题](#)
- [RNN的应用举例——基于RNN的语言模型](#)
 - [向量化](#)
 - [Softmax层](#)
 - [语言模型的训练](#)
 - [交叉熵误差](#)
- [RNN的实现](#)
- [小节](#)
- [参考资料](#)

•

•

◦ 机器学习 7

- [零基础入门深度学习\(7\) - 递归神经网络](#)
- [零基础入门深度学习\(6\) - 长短期记忆网络\(LSTM\)](#)
- [零基础入门深度学习\(5\) - 循环神经网络](#)
- [零基础入门深度学习\(4\) - 卷积神经网络](#)
- [零基础入门深度学习\(3\) - 神经网络和反向传播算法](#)
- [零基础入门深度学习\(2\) - 线性单元和梯度下降](#)
- [零基础入门深度学习\(1\) - 感知器](#)

◦ 深度学习入门 7

- [零基础入门深度学习\(7\) - 递归神经网络](#)
- [零基础入门深度学习\(6\) - 长短期记忆网络\(LSTM\)](#)
- [零基础入门深度学习\(5\) - 循环神经网络](#)
- [零基础入门深度学习\(4\) - 卷积神经网络](#)
- [零基础入门深度学习\(3\) - 神经网络和反向传播算法](#)
- [零基础入门深度学习\(2\) - 线性单元和梯度下降](#)
- [零基础入门深度学习\(1\) - 感知器](#)

◦ [搜索 hanbingtao 的文档](#)

◦ 以下【标签】将用于标记这篇文稿:

•

•

•

- [下载客户端](#)
- [关注开发者](#)
- [报告问题](#), [建议](#)
- [联系我们](#)

•

添加新批注



保存 取消

在作者公开此批注前, 只有你和作者可见。



保存 取消



修改 保存 取消 删除

- 私有
- 公开
- 删除

查看更早的 5 条回复

回复批注

x

通知

取消 确认

• —
• —