# Title
# homework *I* LaTeX

[*] Teacher: Zhou. TA: Yuan Xu

1st 张逸凯

*Department of Computer Science and Technology*

*Nanjing University*

@gmail.com

在我刚开始接触自然语言, 句子序列的学习任务时, 我渐渐认识到这是一般神经网络解决不了的问题, 首先在输入输出层面, 无法做到处理不同长度的句子. 其次一般神经网络无法学习句子中不同位置的词带来的信息. 比如我去中国与我离开中国, 输出层是某个词作为目的地与出发地的概率, 对于上面两句话同样是"中国"这个词的输入, 一般神经网络只能输出同样的结果, 但这显然第一句话中"中国"作为目的地, 第二句话中作为出发地.

RNN的隐含层是有记忆的, 就像我们人在阅读的时候大脑中已经有之前学习过的东西了, 在学习的基础上继续学习, RNN在看见下一个词的时候, 还会把之前输入的作为当前输入的一部分, 共同作用.

第一层的隐含层内激活值会传到下一次输入, 在每一步都会, 直到最后一步. 可以使用零向量来作为第一层的激活值. RNN在处理当前词时, 还考虑之前它所见过的词, 这就是为什么它能学习位置信息的原因.

这里隐含层的参数是共享的. 接下来我将详细推导参数传递的过程.

The hidden layer parameter initialization: $a^{(0)} = \overrightarrow{0}$

First propagation: $a^{(1)} = g_1\left(W_{aa}a^{(0)} + W_{ax}x^{(1)} + b_a\right)$

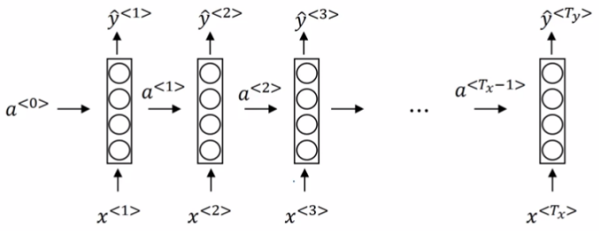
First layer output: $\hat{y}^{(1)} = g\left(W_{y_0}a^{(1)} + b_y\right)$

很容易可以推广到第i层:

$$a^{(t)} = g\left(W_{aa}a^{(t-1)} + W_{ax}x^{(t)} + b_a\right)$$
$$\hat{y}^{(t)} = g\left(W_{ya}a^{(t)} + b_y\right)$$

[*]谢谢老师和助教哥的耐心批改.

这里是指对第一个词的第一次forward propagation. 其中g是激活函数, 一般使用tanh. $x^{<i>}$ 是第i次输入, $\hat{y}^{<i>}$ 是第i次输出. $a$是隐含层参数矩阵, $W$ 是网络的参数矩阵, $b_y$是当前偏置常量.




结合RNN的结构图我们可以更清楚的理解各参数的传导过程. 我在上面故意把$W$矩阵分开来写, 可以更好的分层来看每一词进入网络后的传播过程. 论文里给出的一般都是通过数学处理之后合并的矩阵形式:

$$W \begin{bmatrix} a^{(t-1)} \\ x^{(t)} \end{bmatrix} = [W_{aa} W_{ax}] \begin{bmatrix} a^{(t-1)} \\ x^{(t)} \end{bmatrix} = W_{aa} a^{(t-1)} + W_{ax} x^{(t)}$$

接下来自然是backpropagation through time, 非常有趣的名字, 先来看loss function, 是交叉熵, 它代表了在某个时间步上对一个词预测的损失:

$$\mathcal{L}^{(t)}\left(\hat{y}^{(t)}, y^{(t)}\right) = -y^{(t)} \log \hat{y}^{(t)} - \left(1 - y^{(t)}\right) \log \left(1 - \hat{y}^{(t)}(t)\right)$$

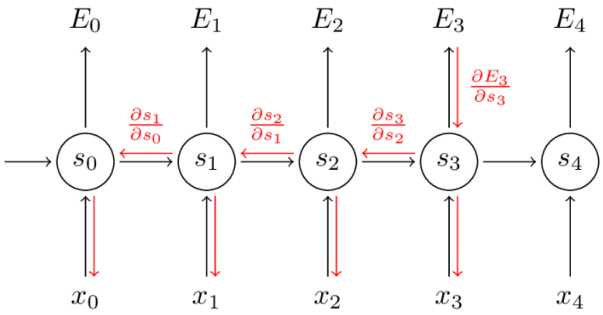
$$\mathcal{L}(\hat{y}, y) = \sum_{t=0}^{T_y} \mathcal{L}^{(t)}\left(\hat{y}^{(t)}, y^{(t)}\right)$$

在前向传播的时候计算loss function, 在其相反方向推导各个参数, 在每一个时间步的推导中, 明确我们需要计算的是相对于参数的误差梯度, 然后通过SGD或者其他计算更新参数. 注意对每一个时间步需要求和:

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_t \frac{\partial \mathcal{L}_t}{\partial W}$$

很自然地运用链式法则:

$$\begin{aligned} \frac{\partial \mathcal{L}_i}{\partial W} &= \frac{\partial \mathcal{L}_i}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial W} \\ &= \frac{\partial \mathcal{L}_i}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial a_i} \frac{\partial a_i}{\partial W} \end{aligned}$$




这里采用PyTorch对实现部分做一个详述, 其实和传统前馈神经网络的传播方式比较, 最大的不同还是在数据流经过网络内部参数的更新方式. RNN不是一次喂进所有数据(微观上), 每一隐含层的结果都会影响下一次输入, RNN 可以被看做是同一神经网络的多次复制，每个神经网络模块会把消息传递给下一个。需要注意的是, 上面所用的例子中RNN都是为了展示方便, 每个隐含层结点被拆成许多个, 其实它们都是一个并且不断循环不断计算.

RNN的结构是有限制的, 主要是对于长距离的记忆效果很差（序列开始的信息在后期保留很少）, 而且序列开头的词汇不能被很好预测, 比如这两句话: "Teddy Roosevelt was a President.", "Teddy bears are on sale". RNN很难确定Teddy是不是一个人名. 在长期信息访问当前处理单元之前，需要按顺序地通过所有之前的单元, 这意味着它很容易遭遇梯度消失问题. 这也为我接下来要读的论文埋下伏笔, 研究LSTM, GRU, 或者更强大的注意力机制, 层级注意力编码方式等. 下面就来说说什么是RNN的梯度消失.

如果有一句很长的话, 意味着在RNN循环较后的结点(较后的词)$\hat{y}^{<i>}$由于距离长, 很难受到开头词的影响, 同样在反向传播过程中, 句子后部区域的偏导数也很难影响到前面. RNN这个问题很容易可以类比到深层神经网络的梯度问题.

接下来学习的是较基础的也是比较有用的解决梯度消失的方法(其他方法有改进矩阵初始化值或者改变激活函数): Gate Recurrent Unit, 它有一个memory cell, 不妨记为$c^{(t)}$, 在简单的GRU中可以认为$c^{(t)} = a^{(t)}$, 刚刚在RNN中提到过, 这就是隐含层: 上一次见过的词或者也可以说成short-term memory, 我们可以得到如下更新迭代过程, 其中包含memory cell和gate的计算:

$$\hat{c}^{(t)} = \tanh\left(W_c\left[\Gamma_{relevance} \times c^{(t-1)}, x^{(t)}\right] + b_u\right)$$
$$\Gamma_{update} = \sigma\left(W_u\left[c^{(t-1)}, x^{(t)}\right] + b_u\right)$$

从gate的含义可以看出, 就是一个filter的作用, 如何体现呢:

$$c^{\langle t\rangle} = \Gamma_u \times \hat{c}^{(t)} + (1 - \Gamma_u) \times c^{\langle t-1\rangle}$$

注意这里的gate是经过 $\sigma$ sigmoid函数处理的, 所以当我们真正需要记忆细胞, 需要保持记忆的时候, gate经过sigmoid函数之后会变得很小, 这样就保持了长期记忆的传递, 在一定程度上解决了梯度消失的问题.

接下来我学习了GRU的一个更强大, 更通用的版本: LSTM. LSTM在序列模型上有着划时代的影响. 对比RNN, 在RNN的每个神经元中, 输入经过隐含层参数(包含之前的输入信息)与激活函数获得下一层输出. 而LSTM就复杂了许多, 在每一个时间步LSTM都接受三种数据: 当前输入, "上一个"神经元的short-term memory以及long-term memory. The short-term memory is referred to as the hidden state, and the long-term memory is known as the cell state. LSTM使用gate(the Input Gate, the Forget Gate, and the Output Gate.)来控制数据到short-term memory和long-term memory, 它们可以有选择地删除无关的数据.

如图我们来慢慢看每一个gate的结构.

首先是Input Gate, 它从当前输入和short-term memory来决定了什么数据要存储在 long-term memory. 如图所示它有两层, 第一层主要是筛选经过的数据, 怎么做呢, 将short-term memory和当前输入喂给sigmoid function. 注意这里不再有$a^{<t>} = c^{<t>}$, 总结一下参数与gate的更新过程:

$$\text{memory cell previous value: } \tilde{c}^{<t>} = \tanh\left(W_c\left[a^{<t-1>}, x^{<t>}\right] + b_c\right.$$

$$\text{update gate: } \Gamma_u = \sigma\left(W_u\left[a^{<t-1>}, x^{<t>}\right] + b_u\right)$$

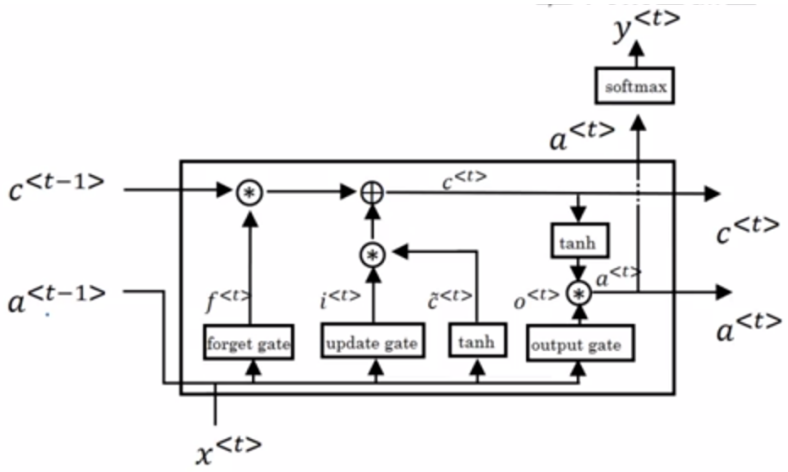
$$\text{forget gate: } \Gamma_f = \sigma\left(W_f\left[a^{<t-1>}, x^{<t>}\right] + b_f\right)$$

$$\text{output gate: } \Gamma_o = \sigma\left(W_o\left[a^{<t-1>}, x^{<t>}\right] + \right\rangle b_o\left.\right)$$

$$\text{memory cell update through: } c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$

最后很自然地有:

$$a^{<t>} = \Gamma_o * \tanh c^{<t>}$$




跟着图可以详细解释一下LSTM的过程: 首先通过forget gate, 来判断什么数据是无用的不相关的, 通过gate的sigmoid function抛弃, 然后根据之前的数据通过 update gate, 刷新一下cell state. 最后再经过output gate返回经过transform后的数据.

word embedding部分:

one hot vector 无法捕捉词与词之间的关系, 没有很好的泛化性能. 所以怎么更好地表达一个词? 很容易想到一个很简单的方法: 使用一些特征, 用这些词在这些特征上的权值来组成特征化向量, 比如与性别相关度, 与食物相关度, 与是否有生命的相关度.