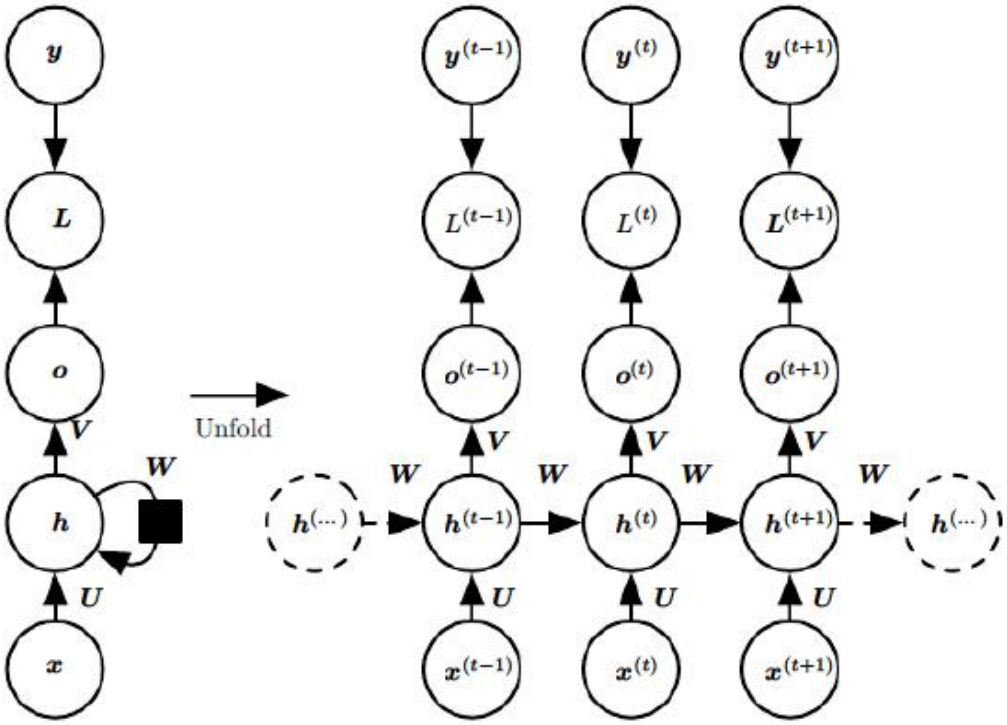
RNN 和 LSTM

RNN 的结构及变体

我们从基础的神经网络中知道，神经网络包含输入层、隐层、输出层，通过激活函数控制输出，层与层之间通过权值连接。激活函数是事先确定好的，那么神经网络模型通过训练“学“到的东西就蕴含在“权值“中。

基础的神经网络只在层与层之间建立了权连接，RNN 最大的不同之处就是在层之间的神经元之间也建立的权连接。如图。



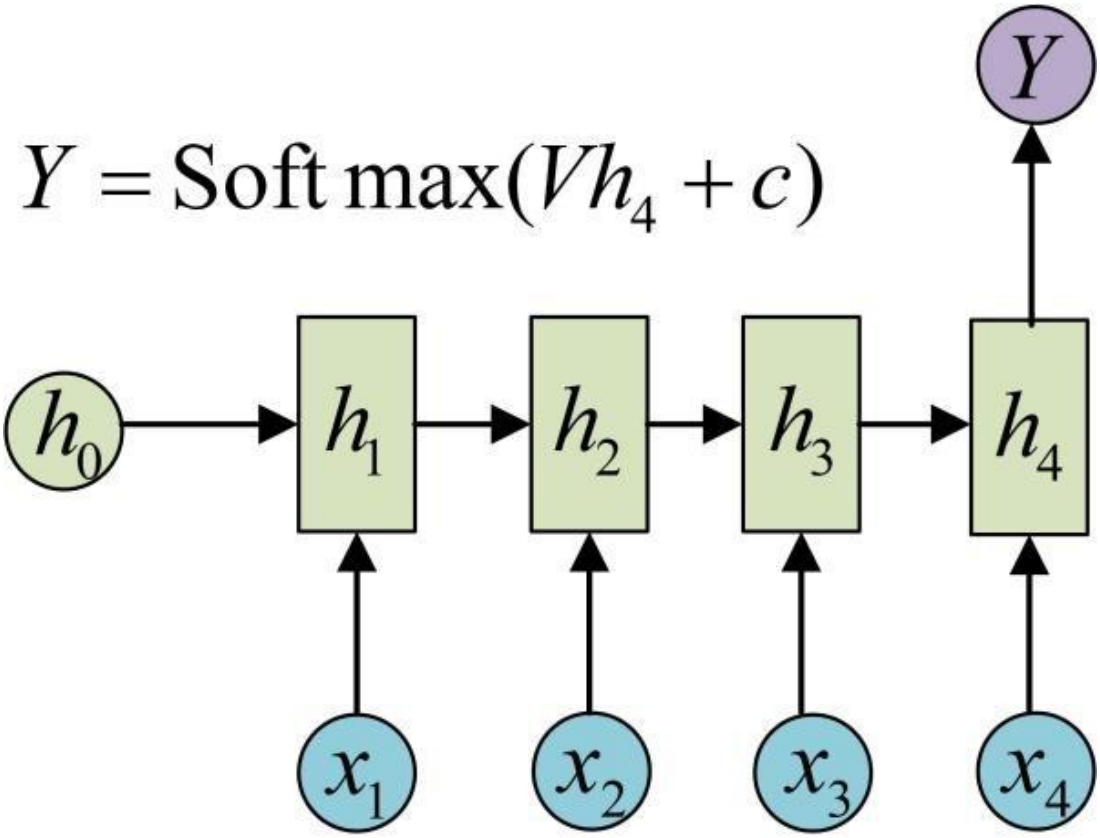
这是一个标准的 RNN 结构图，图中每个箭头代表做一次变换，也就是说箭头连接带有权值。左侧是折叠起来的样子，右侧是展开的样子，左侧中 h 旁边的箭头代表此结构中的“循环“体现在隐层。

在展开结构中我们可以观察到，在标准的 RNN 结构中，隐层的神经元之间也是带有权值的。也就是说，随着序列的不断推进，前面的隐层将会影响后面的隐层。图中 O 代表输出，y 代表样本给出的确定值，L 代表损失函数，我们可以看到，“损失“也是随着序列的推荐而不断积累的。

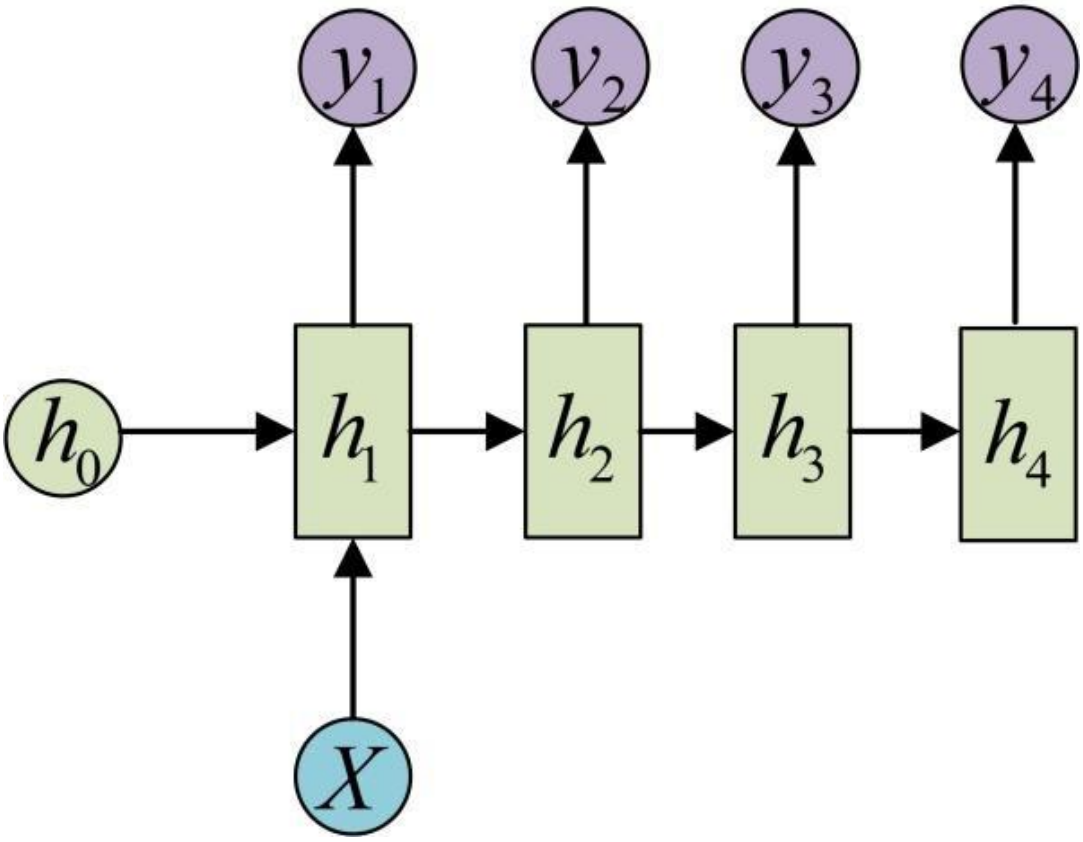
除上述特点之外，标准 RNN 的还有以下特点：

1、权值共享，图中的 W 全是相同的，U 和 V 也一样。

2、每一个输入值都只与它本身的那条路线建立权连接，不会和别的神经元连接。以上是 RNN 的标准结构，然而在实际中这一种结构并不能解决所有问题，例如我们输入为一串文字，输出为分类类别，那么输出就不需要一个序列，只需要单个输出。如图。



同样的，我们有时候还需要单输入但是输出为序列的情况。那么就可以使用如下结构：



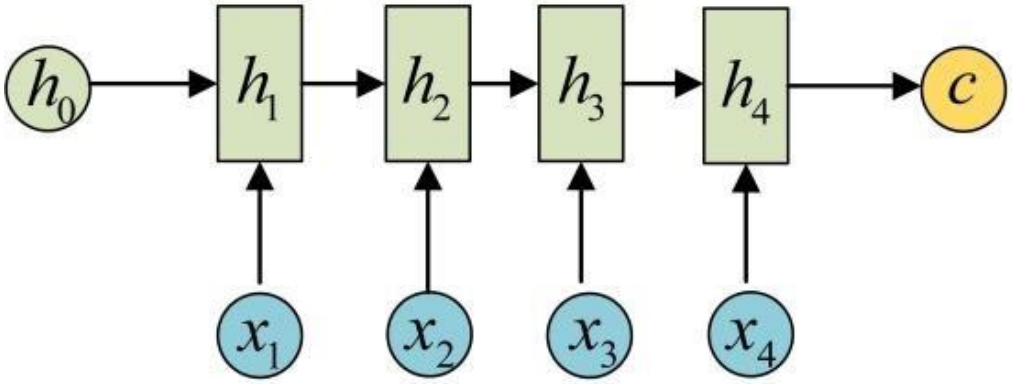
还有一种结构是输入虽是序列，但不随着序列变化，就可以使用如下结构：原始的 N vs N RNN 要求序列等长，然而我们遇到的大部分问题序列都是不等长的，如机器翻译中，源语言和目标语言的句子往往并没有相同的长度。

下面我们来介绍 RNN 最重要的一个变种：N vs M。这种结构又叫 Encoder-Decoder 模型，也可以称之为 Seq2Seq 模型。

$$（1）\ c = h_4$$

$$（2）\ c = q(h_4)$$

$$（3）\ c = q(h_1, h_2, h_3, h_4)$$



从名字就能看出，这个结构的原理是先编码后解码。左侧的 RNN 用来编码得到 c，拿到 c 后再用右侧的 RNN 进行解码。得到 c 有多种方式，最简单的方法就是把 Encoder 的最后一个隐状态赋值给 c，还可以对最后的隐状态做一个变换得到 c，也可以对所有的隐状态做变换。

除了以上这些结构以外 RNN 还有很多种结构，用于应对不同的需求和解决不同的问题。还想继续了解可以看一下下面这个博客，里面又介绍了几种不同的结构。但相同的是循环神经网络除了拥有神经网络都有的一些共性元素之外，它总要在一个地方体现出“循环“，而根据“循环“体现方式的不同和输入输出的变化就形成了多种 RNN 结构。

标准 RNN 的前向输出流程

上面介绍了 RNN 有很多变种，但其数学推导过程其实都是大同小异。这里就介绍一下标准结构的 RNN 的前向传播过程。

再来介绍一下各个符号的含义：x 是输入，h 是隐层单元，o 为输出，L 为损失函数，y 为训练集的标签。这些元素右上角带的 t 代表 t 时刻的状态，其中需要注意的是，因策单元 h 在 t 时刻的表现不仅由此刻的输入决定，还受 t 时刻之前时刻的影响。V、W、U 是权值，同一类型的权连接权值相同。

有了上面的理解，前向传播算法其实非常简单，对于 t 时刻

$$h^{(t)} = \phi(Ux^{(t)} + Wh^{(t-1)} + b)$$

其中ϕ()为激活函数，一般来说会选择 tanh 函数，b 为偏置。t 时刻的输出就更为简单：

$$o^{(t)} = Vh^{(t)} + c$$

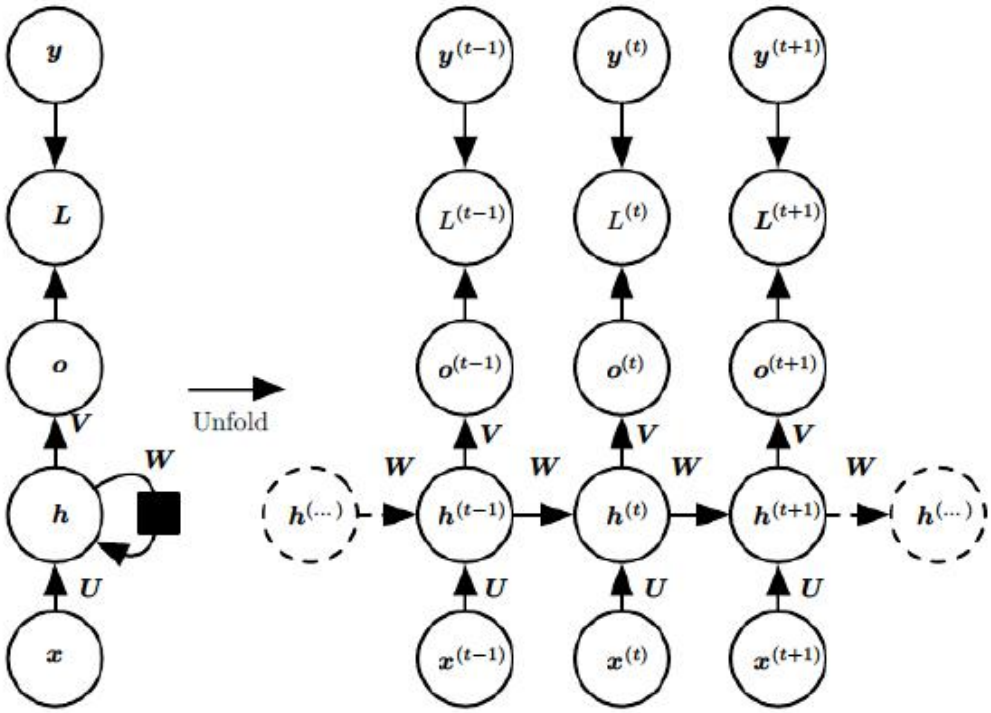
最终模型的预测输出为：

$$\hat{y}^{(t)} = \sigma(o^{(t)})$$

其中 σ () 为激活函数，通常 RNN 用于分类，故这里一般用 softmax 函数。

RNN 的训练方法——BPTT

BPTT（back-propagation through time）算法是常用的训练 RNN 的方法，其实本质还是 BP 算法，只不过 RNN 处理时间序列数据，所以要基于时间反向传播，故叫随时间反向传播。BPTT 的中心思想和 BP 算法相同，沿着需要优化的参数的负梯度方向不断寻找更优的点直至收敛。综上所述，BPTT 算法本质还是 BP 算法，BP 算法本质还是梯度下降法，那么求各个参数的梯度便成了此算法的核心。



再次拿出这个结构图观察，需要寻优的参数有三个，分别是 U、V、W。与 BP

算法不同的是，其中 W 和 U 两个参数的寻优过程需要追溯之前的历史数据，参数 V 相对简单只需关注目前，那么我们就来先求解参数 V 的偏导数。

$$\frac{\partial L^{(t)}}{\partial V} = \frac{\partial L^{(t)}}{\partial o^{(t)}} \cdot \frac{\partial o^{(t)}}{\partial V}$$

这个式子看起来简单但是求解起来很容易出错，因为其中嵌套着激活函数函数，是复合函数的求道过程。

RNN 的损失也是会随着时间累加的，所以不能只求 t 时刻的偏导。

$$L = \sum_{t=1}^{n} L^{(t)}$$

$$\frac{\partial L}{\partial V} = \sum_{t=1}^{n} \frac{\partial L^{(t)}}{\partial o^{(t)}} \cdot \frac{\partial o^{(t)}}{\partial V}$$

W 和 U 的偏导的求解由于需要涉及到历史数据，其偏导求起来相对复杂，我们先假设只有三个时刻，那么在第三个时刻 L 对 W 的偏导数为：

$$\frac{\partial L^{(3)}}{\partial W} = \frac{\partial L^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial W} + \frac{\partial L^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial W} + \frac{\partial L^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial W}$$

相应的，L 在第三个时刻对 U 的偏导数为：

$$\frac{\partial L^{(3)}}{\partial U} = \frac{\partial L^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial U} + \frac{\partial L^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial U} + \frac{\partial L^{(3)}}{\partial o^{(3)}} \frac{\partial o^{(3)}}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial U}$$

可以观察到，在某个时刻的对 W 或是 U 的偏导数，需要追溯这个时刻之前所有时刻的信息，这还仅仅是一个时刻的偏导数，上面说过损失也是会累加的，那么整个损失函数对 W 和 U 的偏导数将会非常繁琐。虽然如此但好在规律还是有迹可循，我们根据上面两个式子可以写出 L 在 t 时刻对 W 和 U 偏导数的通式：

$$\frac{\partial L^{(t)}}{\partial W} = \sum_{k=0}^{t} \frac{\partial L^{(t)}}{\partial o^{(t)}} \frac{\partial o^{(t)}}{\partial h^{(t)}} \left(\prod_{j=k+1}^{t} \frac{\partial h^{(j)}}{\partial h^{(j-1)}} \right) \frac{\partial h^{(k)}}{\partial W}$$

$$\frac{\partial L^{(t)}}{\partial U} = \sum_{k=0}^{t} \frac{\partial L^{(t)}}{\partial o^{(t)}} \frac{\partial o^{(t)}}{\partial h^{(t)}} \left(\prod_{j=k+1}^{t} \frac{\partial h^{(j)}}{\partial h^{(j-1)}} \right) \frac{\partial h^{(k)}}{\partial U}$$

整体的偏导公式就是将其按时刻再一一加起来。

前面说过激活函数是嵌套在里面的，如果我们把激活函数放进去，拿出中间累乘的那部分：

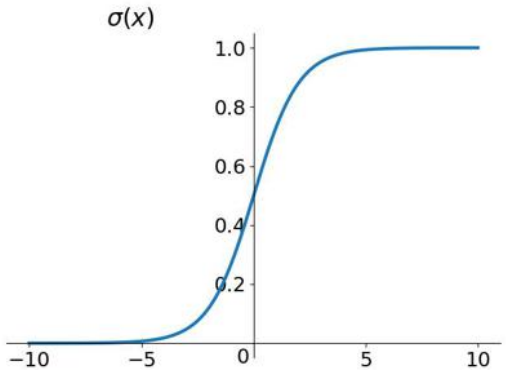
$$\prod_{j=k+1}^{t} \frac{\partial h^j}{\partial h^{j-1}} = \prod_{j=k+1}^{t} tanh^{'} \cdot W_s$$

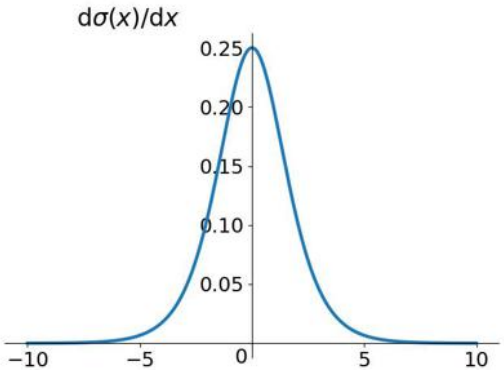
或是：

$$\prod_{j=k+1}^{t} \frac{\partial h^j}{\partial h^{j-1}} = \prod_{j=k+1}^{t} sigmoid^{'} \cdot W_s$$

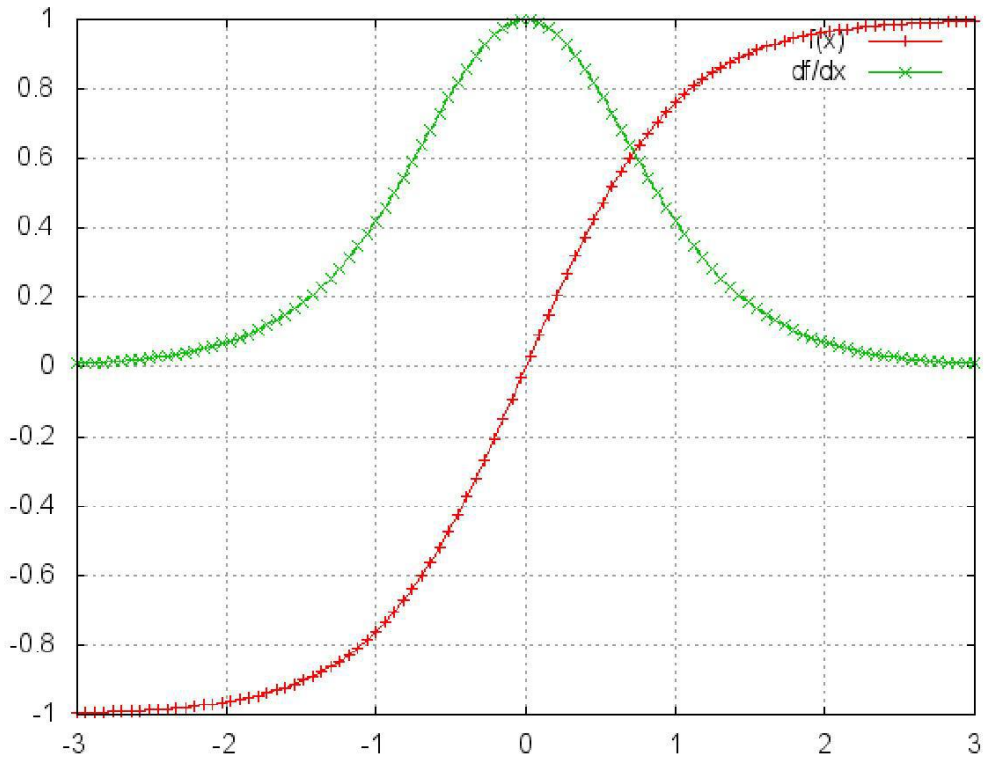
我们会发现累乘会导致激活函数导数的累乘，进而会导致“梯度消失“和“梯度爆炸“现象的发生。至于为什么，我们先来看看这两个激活函数的图像。

下图是 sigmoid 函数的函数图和导数图：





这是 tanh 函数的函数图和导数图。



它们二者是何其的相似，都把输出压缩在了一个范围之内。他们的导数图像也非常相近，我们可以从中观察到，sigmoid 函数的导数范围是(0,0.25]，tach 函数的导数范围是(0,1]，他们的导数最大都不大于 1。

这就会导致一个问题，在上面式子累乘的过程中，如果取 sigmoid 函数作为激活函数的话，那么必然是一堆小数在做乘法，结果就是越乘越小。随着时间序列的不断深入，小数的累乘就会导致梯度越来越小直到接近于 0，这就是“梯度消失“现象。

其实 RNN 的时间序列与深层神经网络很像，在较为深层的神经网络中使用 sigmoid 函数做激活函数也会导致反向传播时梯度消失，梯度消失就意味消失那一层的参数再也不更新，那么那一层隐层就变成了单纯的映射层，毫无意义了，所以在深层神经网络中，有时候多加神经元数量可能会比多加深度好。

你可能会提出异议，RNN 明明与深层神经网络不同，RNN 的参数都是共享的，而且某时刻的梯度是此时刻和之前时刻的累加，即使传不到最深处那浅层也是有梯度的。这当然是对的，但如果我们根据有限层的梯度来更新更多层的共享的参数一定会出现问题的，因为将有限的信息来作为寻优根据必定不会找到所有信息的最优解。

之前说过我们多用 tanh 函数作为激活函数，那 tanh 函数的导数最大也才 1 啊，而且又不可能所有值都取到 1，那相当于还是一堆小数在累乘，还是会出现“梯度消失“，那为什么还要用它做激活函数呢？原因是 tanh 函数相对于 sigmoid 函数来说梯度较大，收敛速度更快且引起梯度消失更慢。

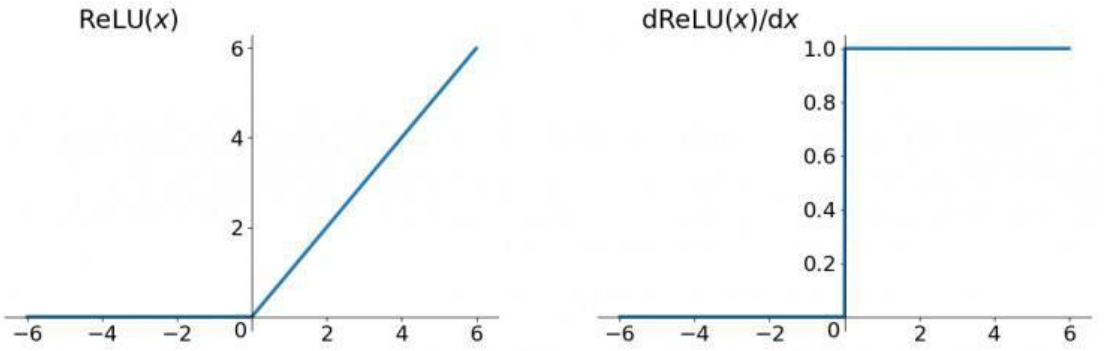
还有一个原因是 sigmoid 函数还有一个缺点，Sigmoid 函数输出不是零中心对称。sigmoid 的输出均大于 0，这就使得输出不是 0 均值，称为偏移现象，这将导致后一层的神经元将上一层输出的非 0 均值的信号作为输入。关于原点对称的输入和中心对称的输出，网络会收敛地更好。

RNN 的特点本来就是能“追根溯源“利用历史数据，现在告诉我可利用的历史数据竟然是有限的，这就令人非常难受，解决“梯度消失“是非常必要的。解决“梯度消失“的方法主要有：

1、选取更好的激活函数

2、改变传播结构

关于第一点，一般选用 ReLU 函数作为激活函数，ReLU 函数的图像为：



ReLU 函数的左侧导数为 0，右侧导数恒为 1，这就避免了“梯度消失“的发生。但恒为 1 的导数容易导致“梯度爆炸“，但设定合适的阈值可以解决这个问题。还有一点就是如果左侧横为 0 的导数有可能导致把神经元学死，不过设置合适的步长（学习旅）也可以有效避免这个问题的发生。

关于第二点，LSTM 结构可以解决这个问题。

总结一下，sigmoid 函数的缺点：

1、导数值范围为(0,0.25]，反向传播时会导致“梯度消失“。tanh 函数导数值范围更大，相对好一点。

2、sigmoid 函数不是 0 中心对称，tanh 函数是，可以使网络收敛的更好。

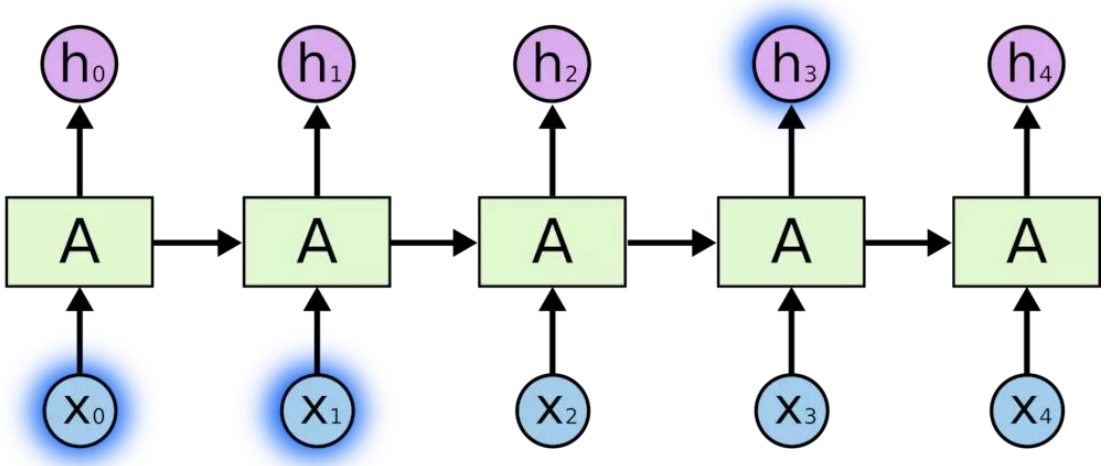
LSTM

下面来了解一下 LSTM（long short-term memory）。长短期记忆网络是 RNN 的一种变体，RNN 由于梯度消失的原因只能有短期记忆，LSTM 网络通过精妙的门控制将短期记忆与长期记忆结合起来，并且一定程度上解决了梯度消失的问题。

长期依赖（Long-Term Dependencies）问题

RNN 的关键点之一就是他们可以用来连接先前的信息到当前的任务上，例如使用过去的视频段来推测对当前段的理解。如果 RNN 可以做到这个，他们就变得非常有用。但是真的可以么？答案是，还有很多依赖因素。

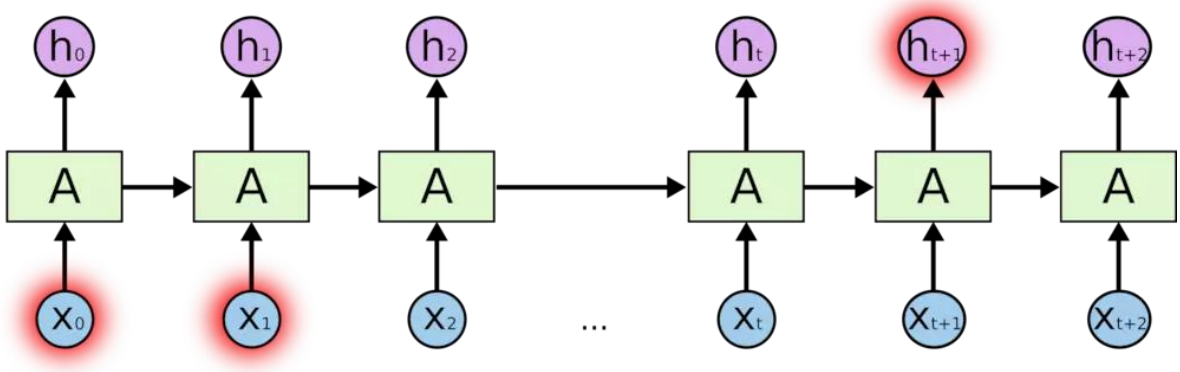
有时候，我们仅仅需要知道先前的信息来执行当前的任务。例如，我们有一个语言模型用来基于先前的词来预测下一个词。如果我们试着预测“the clouds are in the sky” 最后的词，我们并不需要任何其他的上下文 —— 因此下一个词很显然就应该是 sky。在这样的场景中，相关的信息和预测的词位置之间的间隔是非常小的，RNN 可以学会使用先前的信息。



不太长的相关信息和位置间隔

但是同样会有一些更加复杂的场景。假设我们试着去预测“I grew up in France… I speak fluent French”最后的词。当前的信息建议下一个词可能是一种语言的名字，但是如果我们需要弄清楚是什么语言，我们是需要先前提到的离当前位置很远的 France 的上下文的。这说明相关信息和当前预测位置之间的间隔就肯定变得相当的大。

不幸的是，在这个间隔不断增大时，RNN 会丧失学习到连接如此远的信息的能力。



相当长的相关信息和位置间隔

在理论上，RNN 绝对可以处理这样的 长期依赖 问题。人们可以仔细挑选参数来解决这类问题中的最初级形式，但在实践中，RNN 肯定不能够成功学习到这些知识。Bengio, et al. (1994)等人对该问题进行了深入的研究，他们发现一些使训练 RNN 变得非常困难的相当根本的原因。

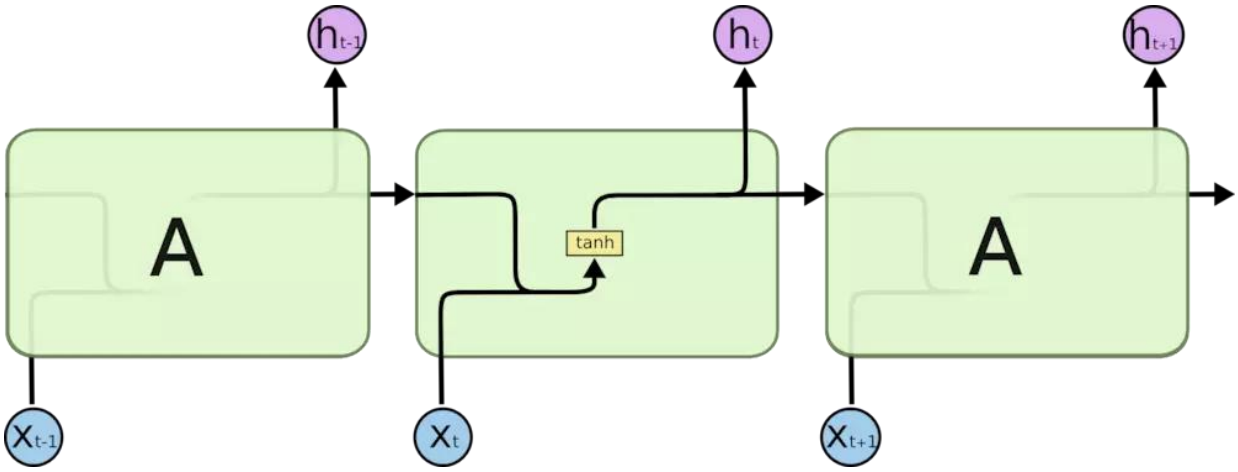
然而，幸运的是，LSTM 并没有这个问题

LSTM 网络

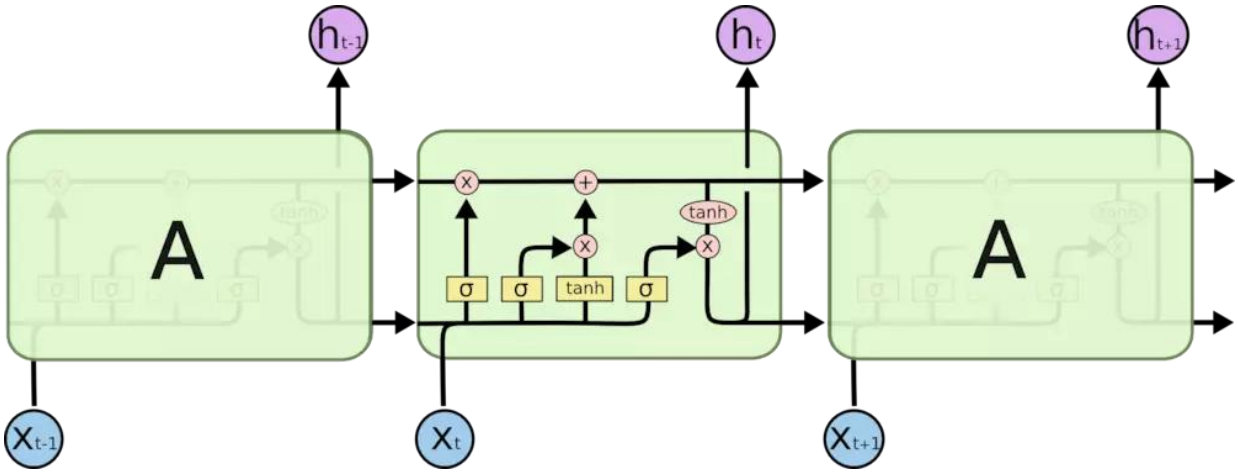
Long Short Term 网络—— 一般就叫做 LSTM ——是一种 RNN 特殊的类型，可以学习长期依赖信息。LSTM 由 Hochreiter & Schmidhuber (1997)提出，并在近期被 Alex Graves 进行了改良和推广。在很多问题，LSTM 都取得相当巨大的成功，并得到了广泛的使用。

LSTM 通过刻意的设计来避免长期依赖问题。记住长期的信息在实践中是 LSTM 的默认行为，而非需要付出很大代价才能获得的能力！

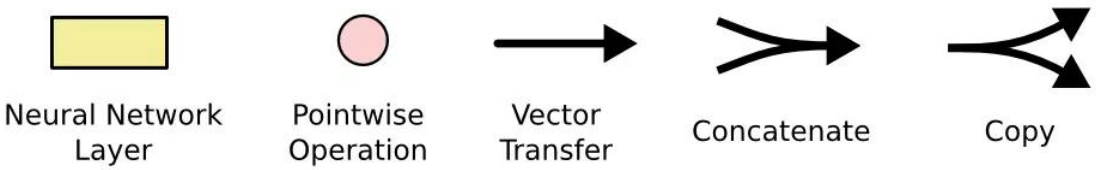
所有 RNN 都具有一种重复神经网络模块的链式的形式。在标准的 RNN 中，这个重复的模块只有一个非常简单的结构，例如一个 tanh 层。



LSTM 同样是这样的结构，但是重复的模块拥有一个不同的结构。不同于 单一神经网络层，整体上除了 h 在随时间流动，细胞状态 c 也在随时间流动，细胞状态 c 就代表着长期记忆。



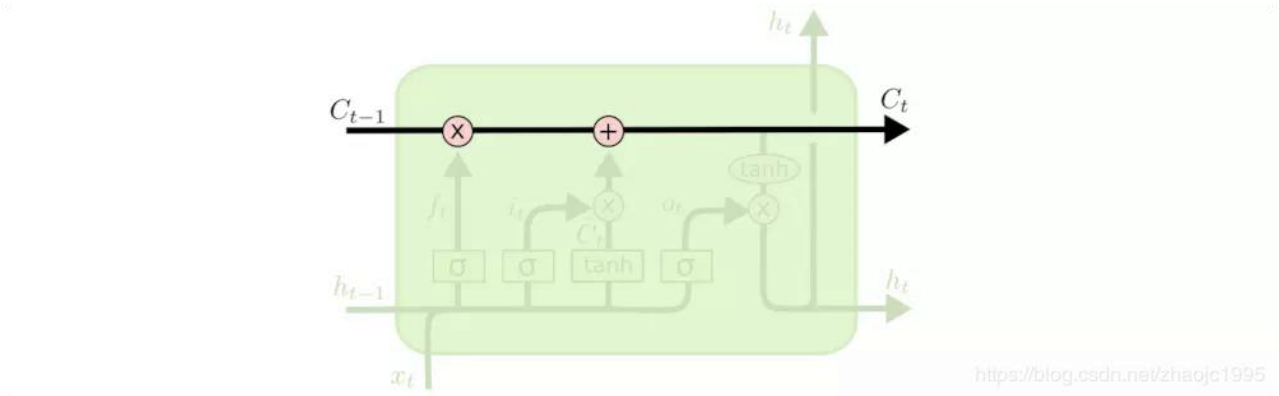
不必担心这里的细节。我们会一步一步地剖析 LSTM 解析图。现在，我们先来熟悉一下图中使用的各种元素的图标。



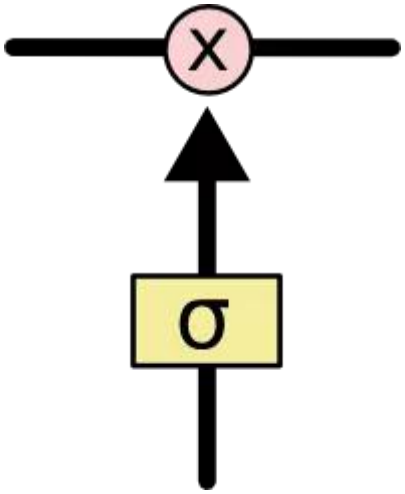
- 黄色的矩形是学习得到的神经网络层
- 粉色的圆形表示一些运算操作，诸如加法乘法
- 黑色的单箭头表示向量的传输
- 两个箭头合成一个表示向量的连接
- 一个箭头分开表示向量的复制

LSTM 的核心思想

LSTM 的关键就是细胞状态，水平线在图上方贯穿运行。细胞状态类似于传送带。直接在整个链上运行，只有一些少量的线性交互。信息在上面流传保持不变会很容易。



LSTM 有通过精心设计的称作为“门”的结构来去除或者增加信息到细胞状态的能力。门是一种让信息选择式通过的方法。他们包含一个 sigmoid 神经网络层和一个 pointwise 乘法操作。



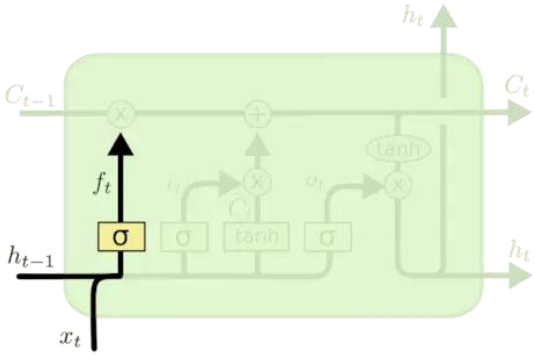
Sigmoid 层输出 0 到 1 之间的数值，描述每个部分有多少量可以通过。0 代表“不许任何量通过”，1 就指“允许任意量通过”！

LSTM 拥有三个门，来保护和控制细胞状态。

逐步理解 LSTM

在我们 LSTM 中的第一步是决定我们会从细胞状态中丢弃什么信息。这个决定通过一个称为遗忘门完成。该门会读取 h_{t-1} 和 x_t 输出一个在 0 到 1 之间的数值给每个在细胞状态 C_{t-1} 中的数字。1 表示“完全保留”，0 表示“完全舍弃”。

让我们回到语言模型的例子中来基于已经看到的预测下一个词。在这个问题中，细胞状态可能包含当前主语的性别，因此正确的代词可以被选择出来。当我们看到新的主语，我们希望忘记旧的主语。



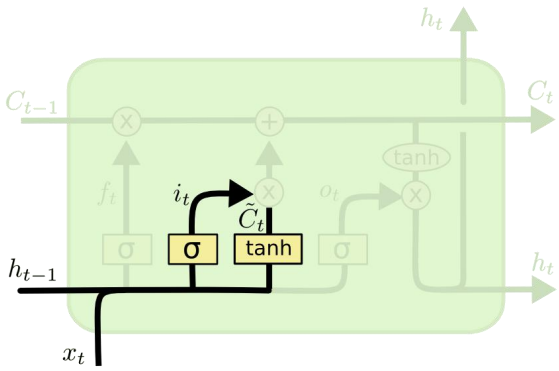
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

这里可以抛出两个问题：这个门怎么做到“遗忘“的呢？怎么理解？既然是遗忘旧的内容，为什么这个门还要接收新的 x_t?

对于第一个问题，“遗忘“可以理解为“之前的内容记住多少“，其精髓在于只能输出（0，1）小数的 sigmoid 函数和粉色圆圈的乘法，LSTM 网络经过学习决定让网络记住以前百分之多少的内容。对于第二个问题就更好理解，决定记住什么遗忘什么，其中新的输入肯定要产生影响。

下一步是确定什么样的新信息被存放在细胞状态中。这里包含两个部分。第一，sigmoid 层称 “输入门层” 决定什么值我们将要更新。然后，一个 tanh 层创建一个新的候选值向量 C_t ，会被加入到状态中。下一步，我们会讲这两个信息来产生对状态的更新。

在我们语言模型的例子中，我们希望增加新的主语的性别到细胞状态中，来替代旧的需要忘记的主语。



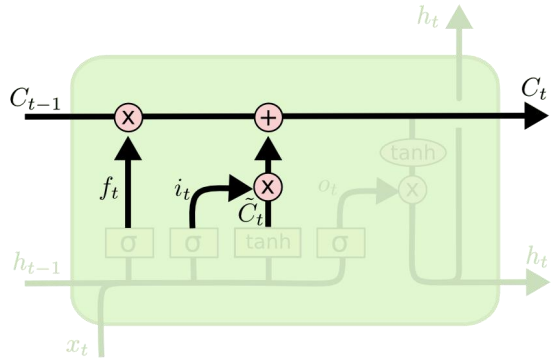
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

现在是更新旧细胞状态的时间了，Ct-1 更新为 Ct。前面的步骤已经决定了将会做什么，我们现在就是实际去完成。

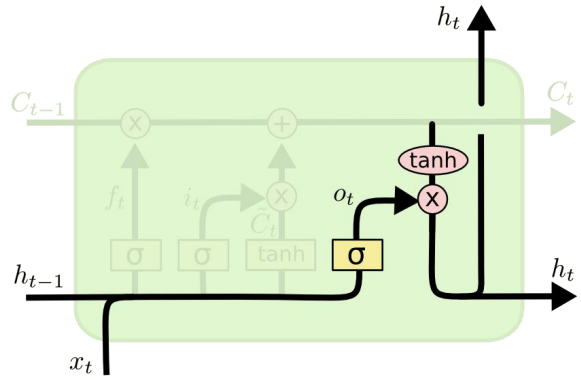
我们把旧状态与 f_t 相乘，丢弃掉我们确定需要丢弃的信息。接着加上相乘，丢弃掉我们确定需要丢弃的信息。接着加上 i_t 与 Ct 相乘，丢弃掉我们确定需要丢弃的信息。根据我们决定更新每个状态的程度进行变化。

有了上面的理解基础输入门，输入门理解起来就简单多了。sigmoid 函数选择更新内容，tanh 函数创建更新候选。



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

最终，我们需要确定输出什么值。这个输出将会基于我们的细胞状态，但是也是一个过滤后的版本。首先，我们运行一个 sigmoid 层来确定细胞状态的哪个部分将输出出去。接着，我们把细胞状态通过 tanh 进行处理（得到一个在 -1 到 1 之间的值）并将它和 sigmoid 门的输出相乘，最终我们仅仅会输出我们确定输出的那部分。



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

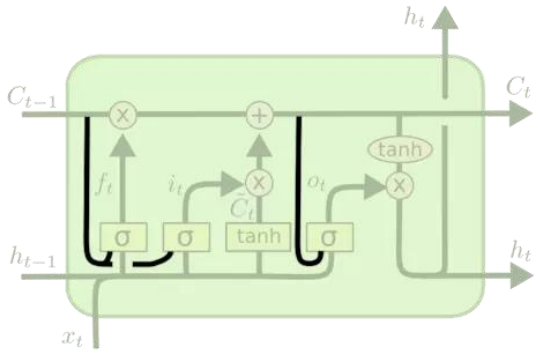
这三个门虽然功能上不同，但在执行任务的操作上是相同的。他们都是使用 sigmoid 函数作为选择工具，tanh 函数作为变换工具，这两个函数结合起来实现三个门的功能。

LSTM 的变体

我们到目前为止都还在介绍正常的 LSTM。但是不是所有的 LSTM 都长成一个样子的。实际上，几乎所有包含 LSTM 的论文都采用了微小的变体。差异非常

小，但是也值得拿出来讲一下。

其中一个流行的 LSTM 变体，就是由 Gers & Schmidhuber (2000) 提出的，增加了 “peephole connection”。是说，我们让 门层 也会接受细胞状态的输入。

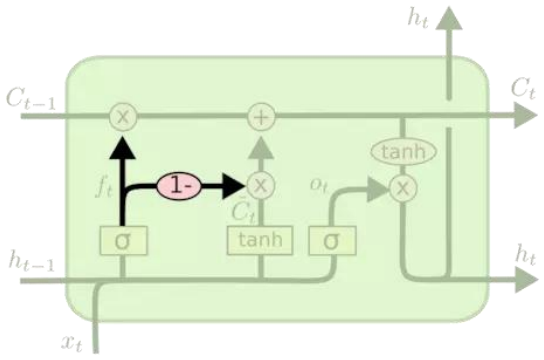


$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

上面的图例中，我们增加了 peephole 到每个门上，但是许多论文会加入部分的 peephole 而非所有都加。

另一个变体是通过使用 coupled 忘记和输入门。不同于之前是分开确定什么忘记和需要添加什么新的信息，这里是一同做出决定。我们仅仅会当我们将要输入在当前位置时忘记。我们仅仅输入新的值到那些我们已经忘记旧的信息的那些状态 。

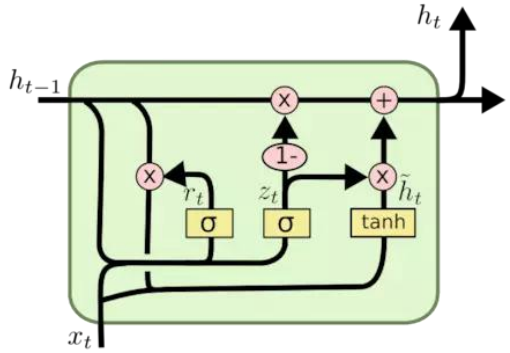
另一个变体是通过使用 coupled 忘记和输入门。不同于之前是分开确定什么忘记和需要添加什么新的信息，这里是一同做出决定。我们仅仅会当我们将要输入在当前位置时忘记。我们仅仅输入新的值到那些我们已经忘记旧的信息的那些状态 。



$$C_t = f_t * C_{t-1} + (\mathbf{1} - \boldsymbol{f_t}) * \tilde{C}_t$$

另一个改动较大的变体是 Gated Recurrent Unit (GRU)，这是由 Cho, et al. (2014) 提出。它将忘记门和输入门合成了一个单一的 更新门。同样还混合了细胞状态和隐藏状态，和其他一些改动。最终的模型比标准的 LSTM 模型要简单，

也是非常流行的变体。



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

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

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

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