

参考文献:

1. Recurrent Neural Networks: <https://cs231n.github.io/rnn/>

Recurrent Neural Networks (RNNs)

One great thing about the RNNs is that they **offer a lot of flexibility** on how we wire up the neural network architecture.

- Normally when we're working with neural networks (**Figure 1**), we are given a fixed sized input vector (red), then we process it with some hidden layers (green), and we produce a fixed sized output vector (blue) as depicted in the leftmost model ("Vanilla" Neural Networks) in **Figure 1**.
- While "Vanilla" Neural Networks receive a single input and produce one **label** for that image, there are tasks where the model produce a sequence of outputs as shown in the one-to-many model in **Figure 1**.

Recurrent Neural Networks allow us to operate over sequences of input, output, or both at the same time.

- An example of **one-to-many model** is **image captioning** where we are given a fixed sized image and produce a sequence of words that describe the content of that image through RNN (second model in Figure 1).
- An example of **many-to-one** task is **action prediction** where we look at a sequence of video frames instead of a single image and produce a **label** of what action was happening in the video as shown in the third model in Figure 1. Another example of many-to-one task is **sentiment classification in NLP** where we are given a sequence of words of a sentence and then classify what sentiment (e.g. positive or negative) that sentence is.
- An example of **many-to-many** task is **video-captioning** where the input is a sequence of video frames and the output is caption that describes what was in the video as shown in the fourth model in Figure 1. Another example of many-to-many task is **machine translation in NLP**, where we can have an RNN that takes a sequence of words of a sentence in English, and then this RNN is asked to produce a sequence of words of a sentence in French.
- There is also a **variation of many-to-many** task as shown in the last model in Figure 1, where the model generates an output at every timestep. An example of this many-to-many task is **video classification on a frame level** where the model classifies every single frame of video with some number of classes. We should note that we don't want this prediction to only be a function of the current timestep (current frame of the video), but also all the timesteps (frames) that have come before this video.

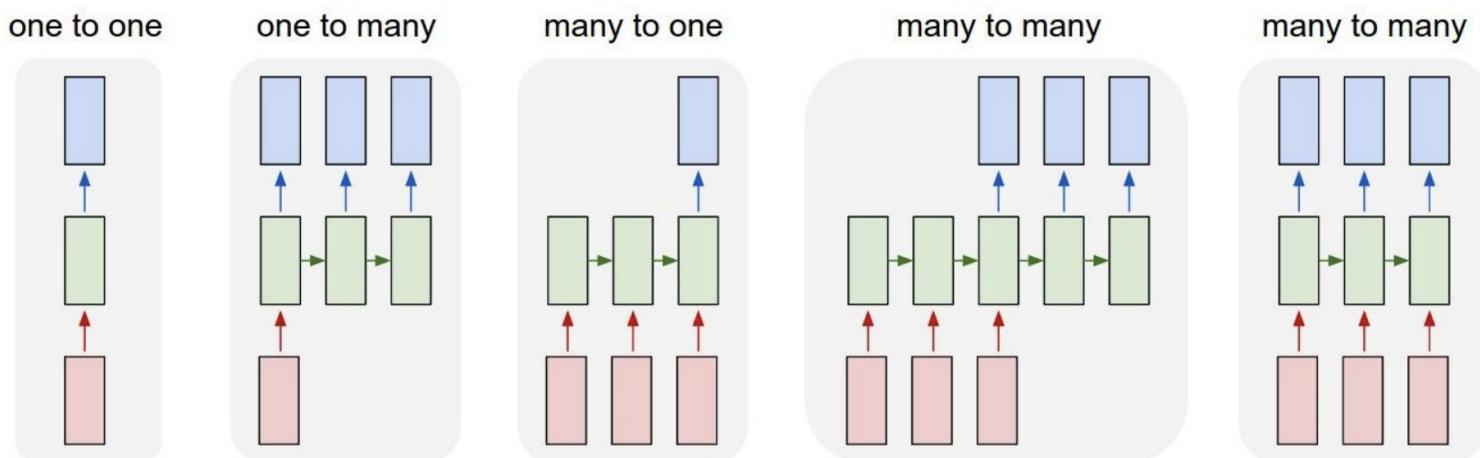


Figure 1. Different (non-exhaustive) types of Recurrent Neural Network architectures. Red boxes are input vectors. Green boxes are hidden layers. Blue boxes are output vectors.

In general, RNNs allow us to wire up an architecture, where the prediction at every single timestep is a function of all the timesteps that have come before.

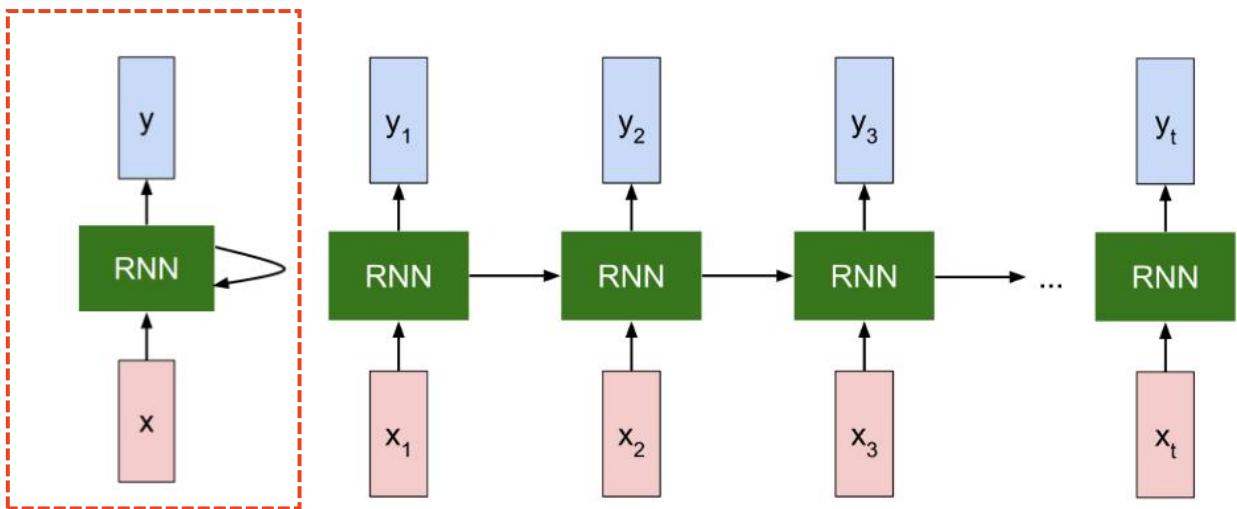


Figure 2. Simplified RNN box (Left) and Unrolled RNN (Right).

$$h_t = f_W(h_{t-1}, x_t)$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$h_t = \tanh(\begin{matrix} \text{dots} \\ \text{dots} \\ \text{dots} \\ \text{dots} \end{matrix} \cdot \begin{matrix} \text{dots} & \text{dots} & \text{dots} \\ \text{dots} & \text{dots} & \text{dots} \\ \text{dots} & \text{dots} & \text{dots} \\ \text{dots} & \text{dots} & \text{dots} \end{matrix} \cdot \begin{matrix} \text{dots} \\ \text{dots} \\ \text{dots} \\ \text{dots} \end{matrix} + \begin{matrix} \text{dots} & \text{dots} & \text{dots} \\ \text{dots} & \text{dots} & \text{dots} \\ \text{dots} & \text{dots} & \text{dots} \\ \text{dots} & \text{dots} & \text{dots} \end{matrix} \cdot \begin{matrix} \text{dots} \\ \text{dots} \\ \text{dots} \\ \text{dots} \end{matrix})$$

h_t W_{hh} h_{t-1} W_{xh} x_t

$$y_t = W_{hy}h_t$$

$$y_t = \begin{matrix} \text{dots} \\ \text{dots} \end{matrix} = \begin{matrix} \text{dots} & \text{dots} & \text{dots} & \text{dots} \\ \text{dots} & \text{dots} & \text{dots} & \text{dots} \end{matrix} \cdot \begin{matrix} \text{dots} \\ \text{dots} \\ \text{dots} \\ \text{dots} \end{matrix}$$

y_t W_{hy} h_t

参考文献:

1. Recurrent Neural Networks: <https://cs231n.github.io/rnn/>

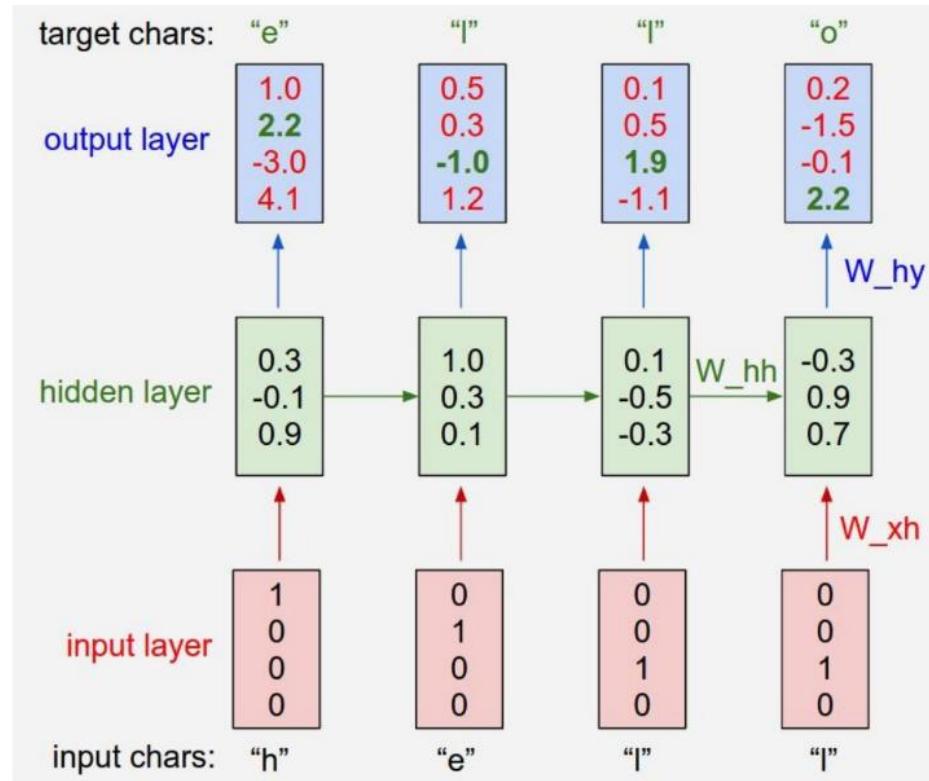


Figure 3. Simplified Character-level Language Model RNN.

$$\begin{bmatrix} 0.3 \\ -0.1 \\ 0.9 \end{bmatrix} = f_W(W_{hh}) \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + W_{xh} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (1)$$

$$\begin{bmatrix} 1.0 \\ 0.3 \\ 0.1 \end{bmatrix} = f_W(W_{hh}) \begin{bmatrix} 0.3 \\ -0.1 \\ 0.9 \end{bmatrix} + W_{xh} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad (2)$$

$$\begin{bmatrix} 0.1 \\ -0.5 \\ -0.3 \end{bmatrix} = f_W(W_{hh}) \begin{bmatrix} 1.0 \\ 0.3 \\ 0.1 \end{bmatrix} + W_{xh} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad (3)$$

$$\begin{bmatrix} -0.3 \\ 0.9 \\ 0.7 \end{bmatrix} = f_W(W_{hh}) \begin{bmatrix} 0.1 \\ -0.5 \\ -0.3 \end{bmatrix} + W_{xh} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad (4)$$

参考文献:

1. Recurrent Neural Networks: <https://cs231n.github.io/rnn/>

Multilayer RNNs

So far we have only shown RNNs with just one layer. However, we're not limited to only a single layer architectures. One of the ways, RNNs are used today is in more complex manner. RNNs can be stacked together in multiple layers, which gives more depth, and empirically deeper architectures tend to work better (Figure 4).

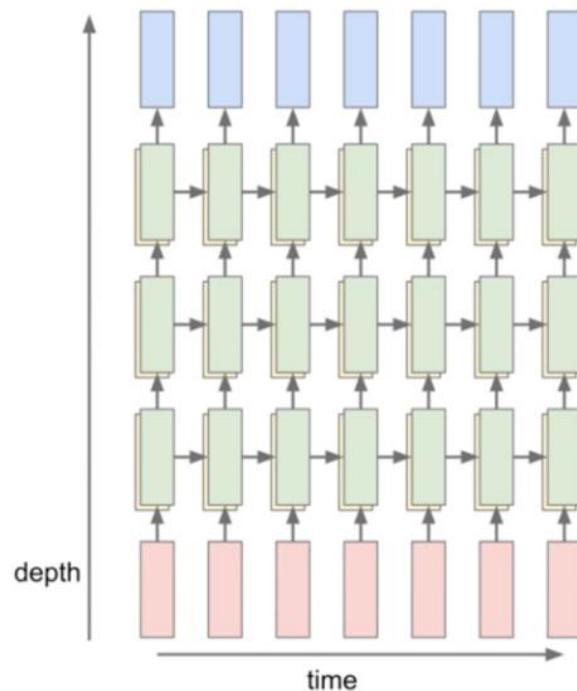


Figure 4. Multilayer RNN example.

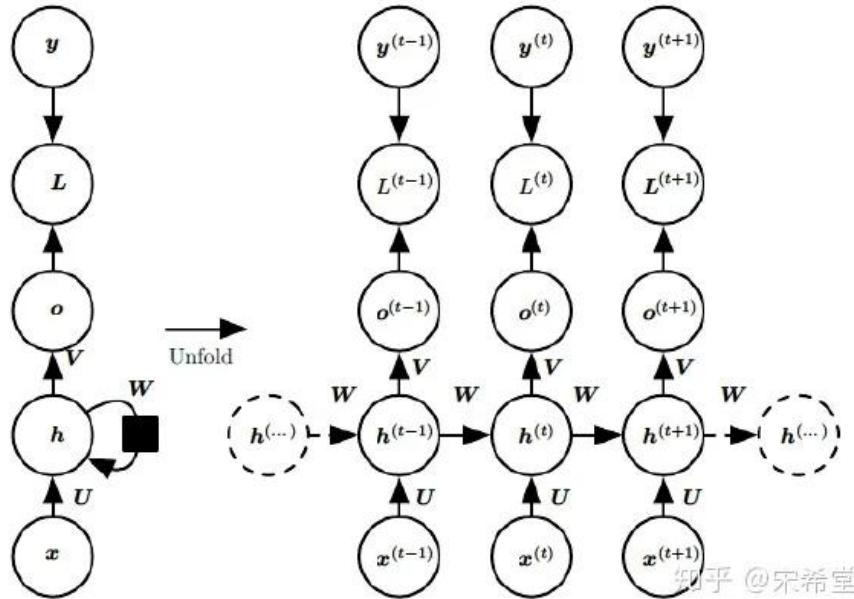
For example, in Figure 4, there are three separate RNNs each with their own set of weights. Three RNNs are stacked on top of each other, so the input of the second RNN (second RNN layer in Figure 4) is the vector of the hidden state vector of the first RNN (first RNN layer in Figure 4). All stacked RNNs are trained jointly, and the diagram in Figure 4 represents one computational graph.

参考文献：

1. 一文看尽RNN（循环神经网络）：<https://zhuanlan.zhihu.com/p/112998607>

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 代表 t 时刻的状态，其中需要注意的是，隐藏单元 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：

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

其中 $\phi()$ 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：

$$o^{(t)} = Vh^{(t)} + c \quad (2)$$

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。

公式 (1) 是隐藏层的计算公式，它是循环层。公式 (2) 是输出层的计算公式，输出层是一个全连接层

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 。

参考文献:

1. 一文看尽RNN（循环神经网络）：<https://zhuanlan.zhihu.com/p/112998607>

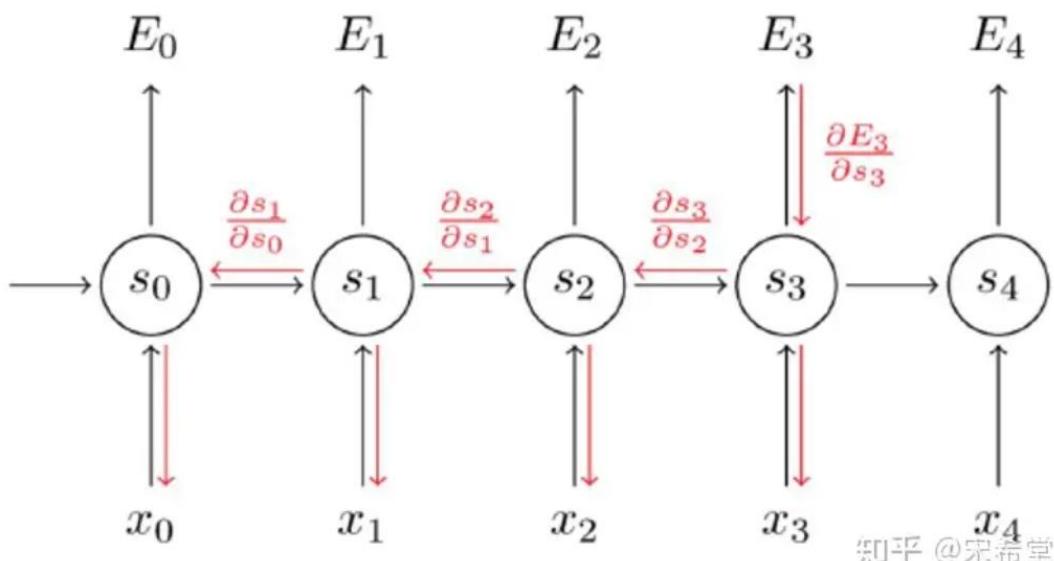
如果反复把公式（1）带入到公式（2），先去掉偏置，我们将得到：

$$\begin{aligned}o^{(t)} &= g(Vh^{(t)}) \\&= Vf(Ux^{(t)} + Wh^{(t-1)}) \\&= Vf(Ux^{(t)} + Wf(Ux^{(t-1)} + Wh^{(t-2)})) \\&= Vf(Ux^{(t)} + Wf(Ux^{(t-1)} + Wf(Ux^{(t-2)} + Wh^{(t-3)}))) \\&= Vf(Ux^{(t)} + Wf(Ux^{(t-1)} + Wf(Ux^{(t-2)} + Wf(Ux^{(t-3)} + \dots))))\end{aligned}$$

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。

RNN的训练方法——BPTT

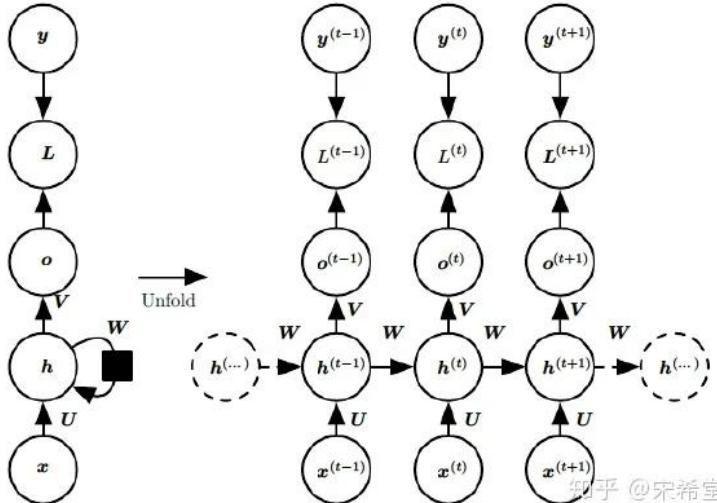
BPTT (back-propagation through time) 算法是常用的训练RNN的方法，其实本质还是BP算法，只不过RNN处理时间序列数据，所以要基于时间反向传播，故叫随时间反向传播。BPTT的中心思想和BP算法相同，沿着需要优化的参数的负梯度方向不断寻找更优的点直至收敛。综上所述，BPTT算法本质还是BP算法，BP算法本质还是梯度下降法，那么求各个参数的梯度便成了此算法的核心。



知乎 @宋希望

参考文献：

1. 一文看尽RNN（循环神经网络）：<https://zhuanlan.zhihu.com/p/112998607>



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：

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

参考文献:

1. 一文看尽RNN（循环神经网络）：<https://zhuanlan.zhihu.com/p/112998607>

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：

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

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。

参考文献：

1. 一文看尽RNN（循环神经网络）：<https://zhuanlan.zhihu.com/p/112998607>

BRNN (双向循环神经网络)

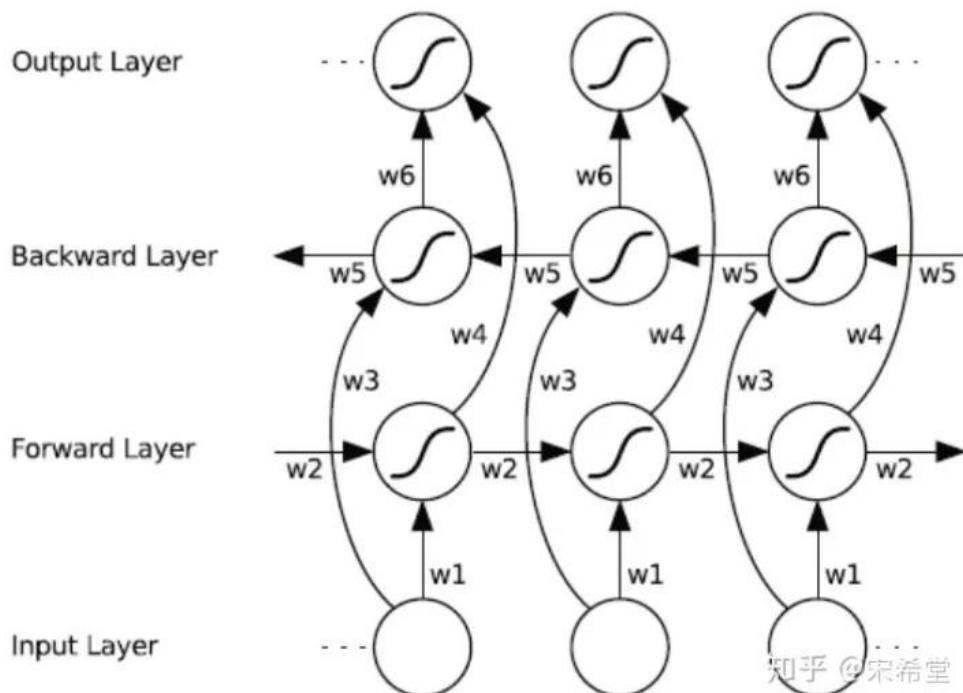
在RNN (Bi-directional Recurrent Neural Network) 中只考虑了预测词前面的词，即只考虑了上下文中“上文”，并没有考虑该词后面的内容。这可能会错过了一些重要的信息，使得预测的内容不够准确。如果能像访问过去的上下文信息一样，访问未来的上下文，这样对于许多序列标注任务是非常有益的。

双向RNN，即可以从过去的时间点获取记忆，又可以从未来的时间点获取信息。为什么要获取未来的信息呢？

判断下面句子中Teddy是否是人名，如果只从前面两个词是无法得知Teddy是否是人名，如果能有后面的信息就很好判断了，这就需要用的双向循环神经网络。

He said, “Teddy bears are on sale!”
He said, “Teddy Roosevelt was a great President!”

双向循环神经网络 (BRNN) 的基本思想是提出每一个训练序列向前和向后分别是两个循环神经网络 (RNN)，而且这两个都连接着一个输出层。这个结构提供给输出层输入序列中每一个点的完整的过去和未来的上下文信息。下图展示的是一个沿着时间展开的双向循环神经网络。六个独特的权值在每一个时步被重复的利用，六个权值分别对应：输入到向前和向后隐含层 (w_1, w_3)，隐含层到隐含层自己 (w_2, w_5)，向前和向后隐含层到输出层 (w_4, w_6)。值得注意的是：向前和向后隐含层之间没有信息流，这保证了展开图是非循环的。



至于网络单元到底是标准的 RNN 还是 GRU 或者是 LSTM 是没有关系的，都可以使用。

参考文献:

1. 一文看尽RNN（循环神经网络）：<https://zhuanlan.zhihu.com/p/112998607>

对于整个双向循环神经网络（BRNN）的计算过程如下：

向前推算（Forward pass）：

对于双向循环神经网络（BRNN）的隐含层，向前推算跟单向的循环神经网络（RNN）一样，除了输入序列对于两个隐含层是相反方向的，输出层直到两个隐含层处理完所有的全部输入序列才更新：

```
for t = 1 to T do
    Forward pass for the forward hidden layer, storing activations at each
    timestep
for t = T to 1 do
    Forward pass for the backward hidden layer, storing activations at each
    timestep
for all t, in any order do
    Forward pass for the output layer, using the stored activations from both
    hidden layers
```

知乎 @宋希望

向后推算（Backward pass）：

双向循环神经网络（BRNN）的向后推算与标准的循环神经网络（RNN）通过时间反向传播相似，除了所有的输出层 δ 项首先被计算，然后返回给两个不同方向的隐含层：

```
for all t, in any order do
    Backward pass for the output layer, storing  $\delta$  terms at each timestep
for t = T to 1 do
    BPTT backward pass for the forward hidden layer, using the stored  $\delta$ 
    terms from the output layer
for t = 1 to T do
    BPTT backward pass for the backward hidden layer, using the stored  $\delta$ 
    terms from the output layer
```

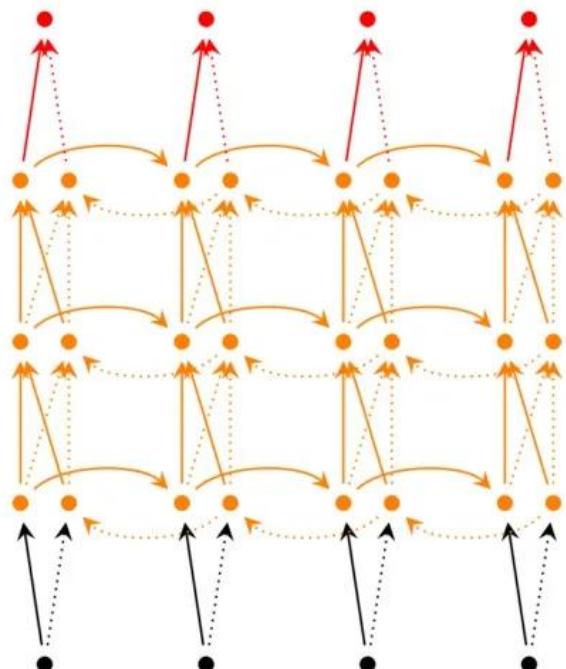
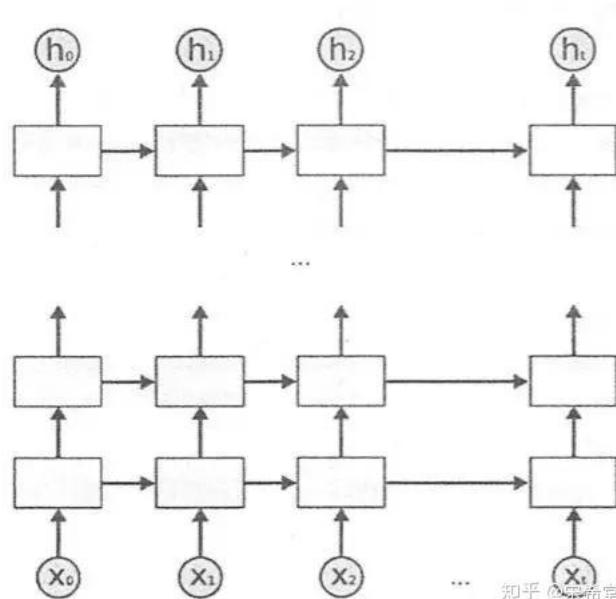
知乎 @宋希望

参考文献：

1. 一文看尽RNN（循环神经网络）：<https://zhuanlan.zhihu.com/p/112998607>

深层RNN(DRNN)

深层RNN网络是在RNN模型多了几个隐藏层，是因为考虑到当信息量太大的时候一次性保存不下所有重要信息，通过多个隐藏层可以保存更多的信息，正如我们看电视剧的时候也可能重复看同一集记住更多关键剧情。同样的，我们也可以在双向RNN模型基础上加多几层隐藏层得到**深层双向RNN模型**。



知乎 @宋希望

注：每一层循环体中参数是共享的，但是不同层之间的权重矩阵是不同的。

至于网络单元到底是标准的RNN还是GRU或者是LSTM是没有关系的，都可以使用。

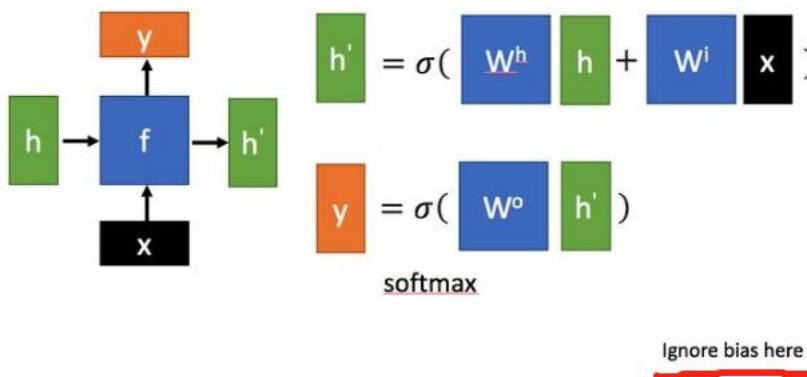
参考文献:

- GRU是什么? RNN、LSTM分别是什么? : https://blog.csdn.net/Frank_LJiang/article/details/103142557

循环神经网络 (Recurrent Neural Network, RNN) 是一种用于**处理序列数据的神经网络**。相比一般的神经网络来说, 它能够处理序列变化的数据。比如某个单词的意思会受上文提到的内容的影响, RNN就能够很好地解决这类问题。其主要形式如下图所示 (台大李宏毅教授的PPT) :

Naïve RNN

- Given function f: $h' = f(h, x)$



- x 为当前时刻的输入特征,
- h 表示上一时刻存储的状态信息。
- y 为当前节点状态下的输出,
- h' 为传递到下一时刻存储的状态信息。

通过上图的公式可以看到, 状态信息 h' 与当前时刻的输入特征 x 和上一时刻的状态信息 h 的值都相关。而 y 则常常使用 h' 投入到一个**线性层** (主要是进行**维度映射**) , 然后使用 **softmax** 进行分类得到需要的数据。对这里的 y 如何通过 h' 计算得到**往往看具体模型的使用方式**。

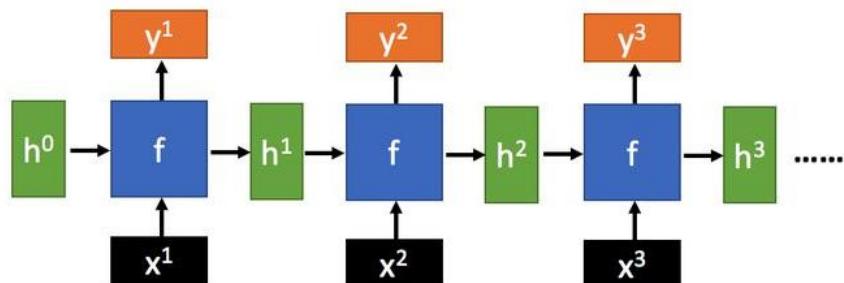
- 第一个激活函数通常用 **tanh**, 第二个激活函数用 **softmax**。
- 前向传播时**: 记忆体内存储的状态信息 h_t , 在每个时刻都被刷新, 三个参数矩阵 w^h, w^i, w^o 自始至终都是固定不变的。
- 反向传播时**: 三个参数矩阵 w^h, w^i, w^o 被梯度下降法更新。

通过序列形式的输入，我们能够得到如下形式的RNN，图中一共有三个循环核（循环计算层）：

Recurrent Neural Network

- Given function $f: h', y = f(h, x)$

h and h' are vectors with the same dimension



No matter how long the input/output sequence is,
we only need one function f

入RNN时，x_train维度：

【送入样本数，循环核时间展开步数，每个时间步输入特征个数】



在 TensorFlow 中，对于 RNN 输入必须是三个维度，分别是样本的总数，循环核时间展开步数（循环核个数），每个时间步输入的特征个数。如上图所示。

下面给出一个例子：

```
1 input_word = "abcde"
2 w_to_id = {'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4} # 单词映射到数值id的词典
3 id_to_onehot = {0: [1., 0., 0., 0., 0.], 1: [0., 1., 0., 0., 0.], 2: [0., 0., 1., 0., 0.], 3: [0., 0., 0., 1., 0.],
4 4: [0., 0., 0., 0., 1.]} # id编码为one-hot
5
6 x_train = [id_to_onehot[w_to_id['a']], id_to_onehot[w_to_id['b']], id_to_onehot[w_to_id['c']],
7 id_to_onehot[w_to_id['d']], id_to_onehot[w_to_id['e']]]
8 y_train = [w_to_id['b'], w_to_id['c'], w_to_id['d'], w_to_id['e'], w_to_id['a']]
```

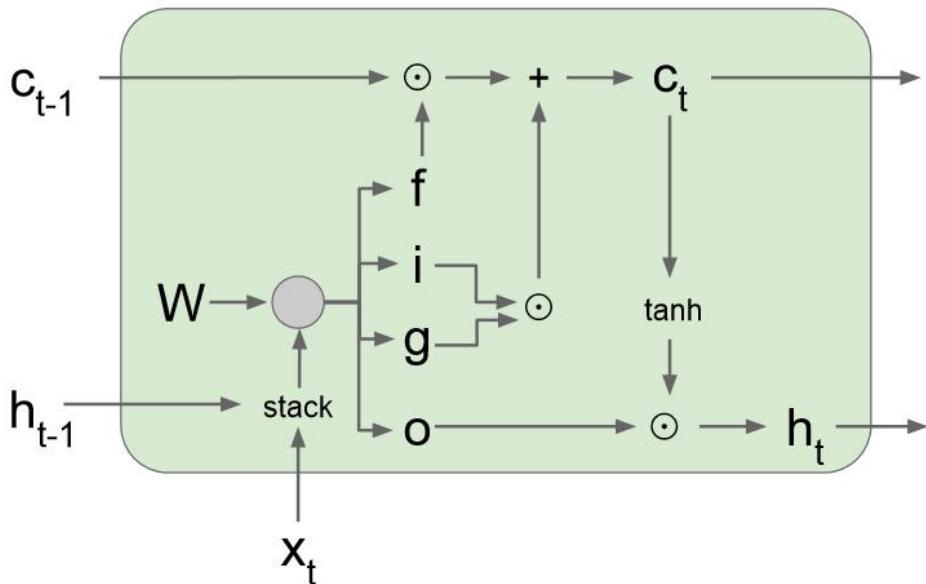
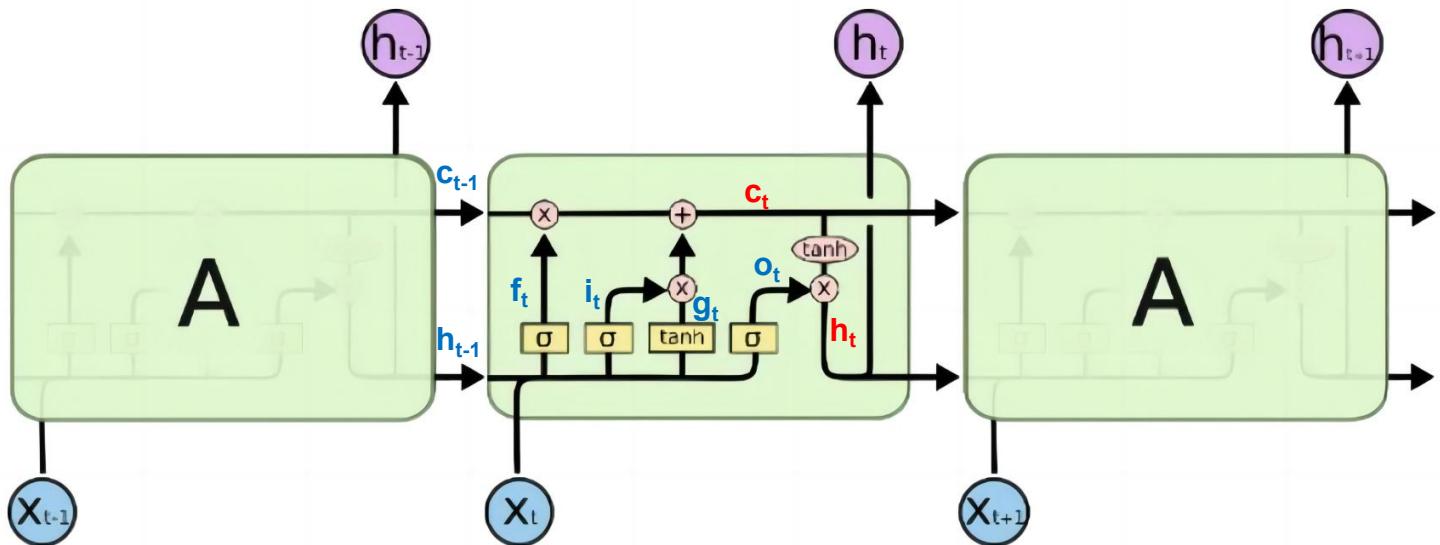
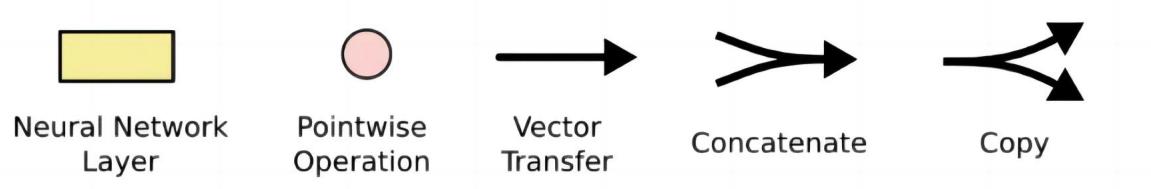
```
1 # 使x_train符合SimpleRNN输入要求：[送入样本数，循环核时间展开步数，每个时间步输入特征个数]。
2 # 此处整个数据集送入，送入样本数为len(x_train)，输入1个字母出结果，循环核时间展开步数为1；表示为独热码有5个输入特征，每个时间步
3 x_train = np.reshape(x_train, (len(x_train), 1, 5))
4 y_train = np.array(y_train)
5
6 # 模型 循环核时间展开步数为1
7 model = tf.keras.Sequential([
8     SimpleRNN(3),
9     Dense(5, activation='softmax')
10])
```

LSTM

参考文献:

1. RNN详解(Recurrent Neural Network): <https://blog.csdn.net/bestrivern/article/details/90723524>

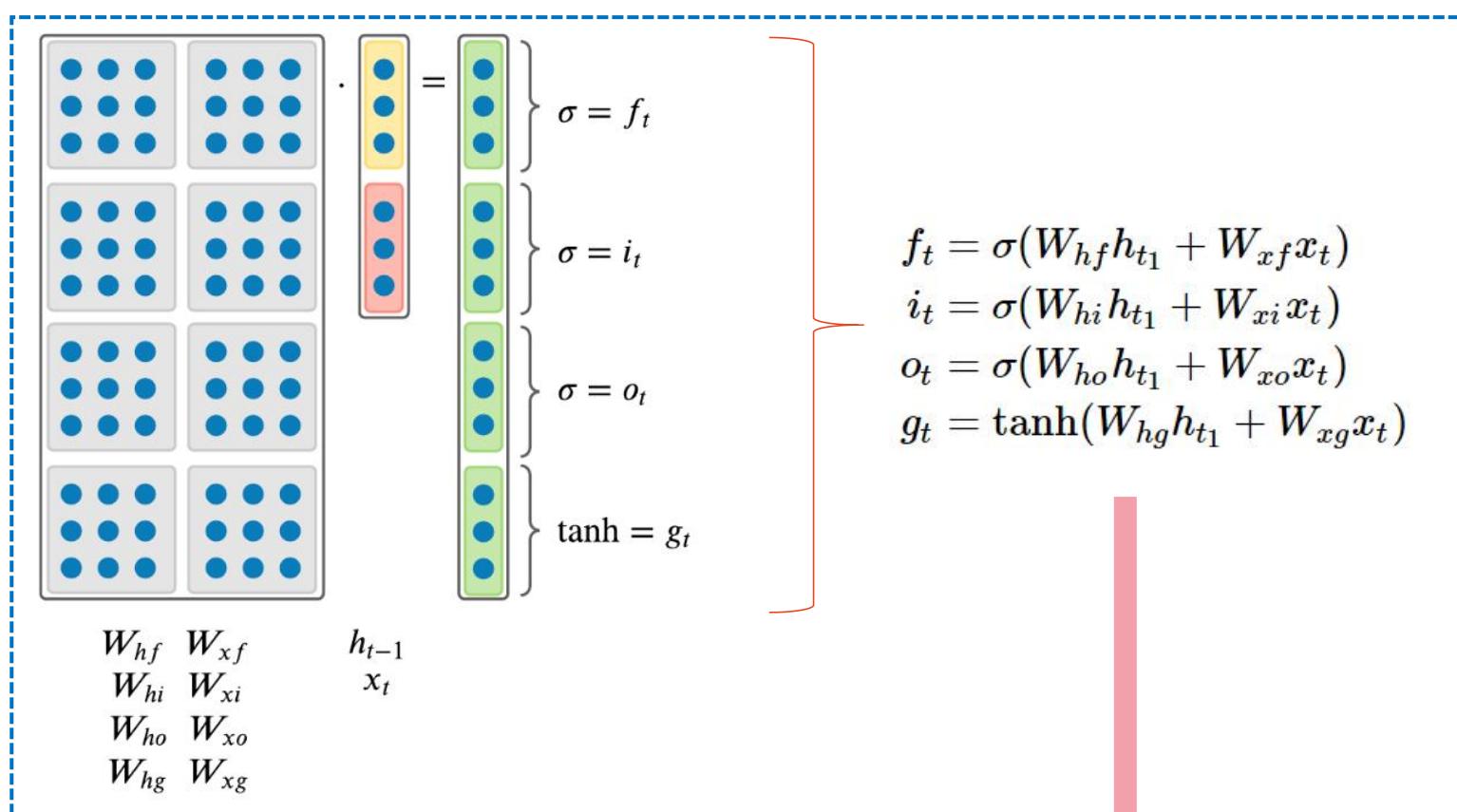
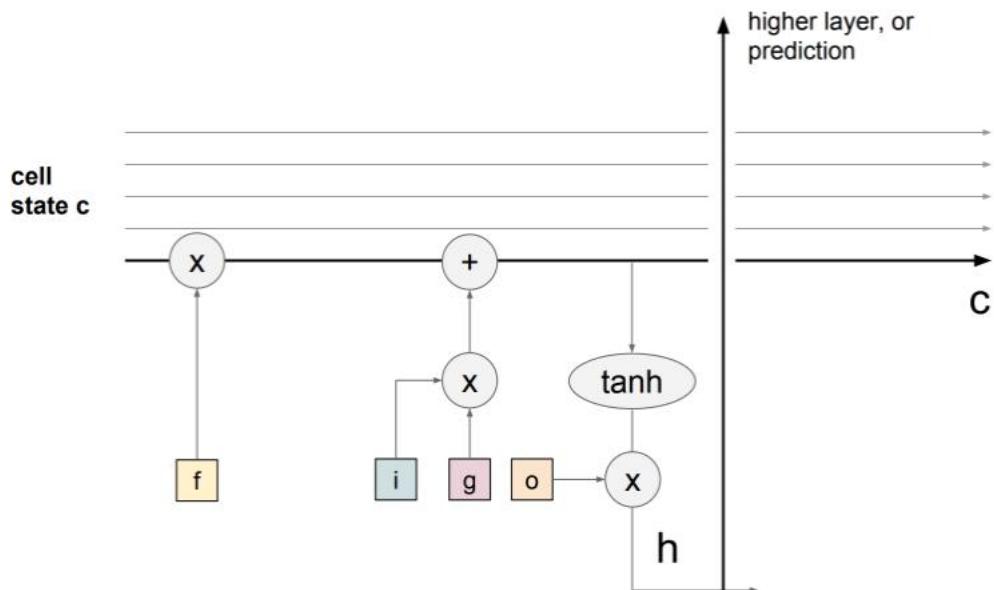
LSTM 中的重复模块包含四个交互的层。



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$
$$c_t = f \odot c_{t-1} + i \odot g$$
$$h_t = o \odot \tanh(c_t)$$

参考文献:

1. Recurrent Neural Networks: <https://cs231n.github.io/rnn/>



The equations show the iterative update of the cell state c_t and hidden state h_t over time t . The cell state is updated as the sum of the previous cell state c_{t-1} and the product of the input gate i_t and the cell candidate g_t , scaled by the forget gate f_t . The hidden state is updated as the product of the output gate o_t and the tanh of the cell state c_t .

$c_t = f_t \odot c_{t-1} + i_t \odot g_t$
 $h_t = o_t \odot \text{tanh}(c_t)$

参考文献:

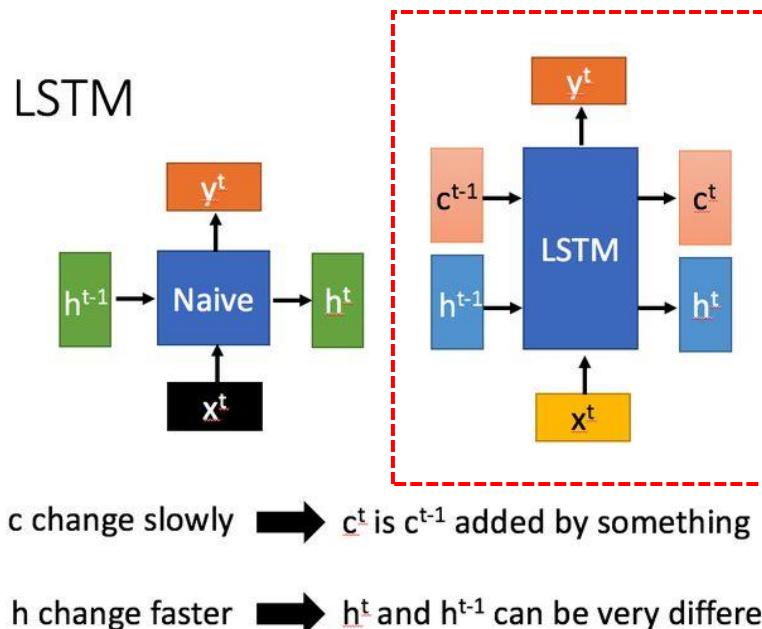
1. GRU是什么? RNN、LSTM分别是什么? : https://blog.csdn.net/Frank_LJiang/article/details/103142557

LSTM

长短期记忆 (Long short-term memory, LSTM) 是一种特殊的 RNN，主要是为了解决长序列训练过程中的**梯度消失**和**梯度爆炸**问题。简单来说，就是相比普通的RNN，LSTM能够在更长的序列中有更好的表现。

传统的循环神经网络RNN可以通过记忆体实现短期记忆进行连续数据的预测，但是当连续数据的序列变长时，会使展开时间步过长，在反向传播更新参数时，梯度要按照时间步连续相乘，很容易导致梯度消失，所以 LSTM 就出现了。

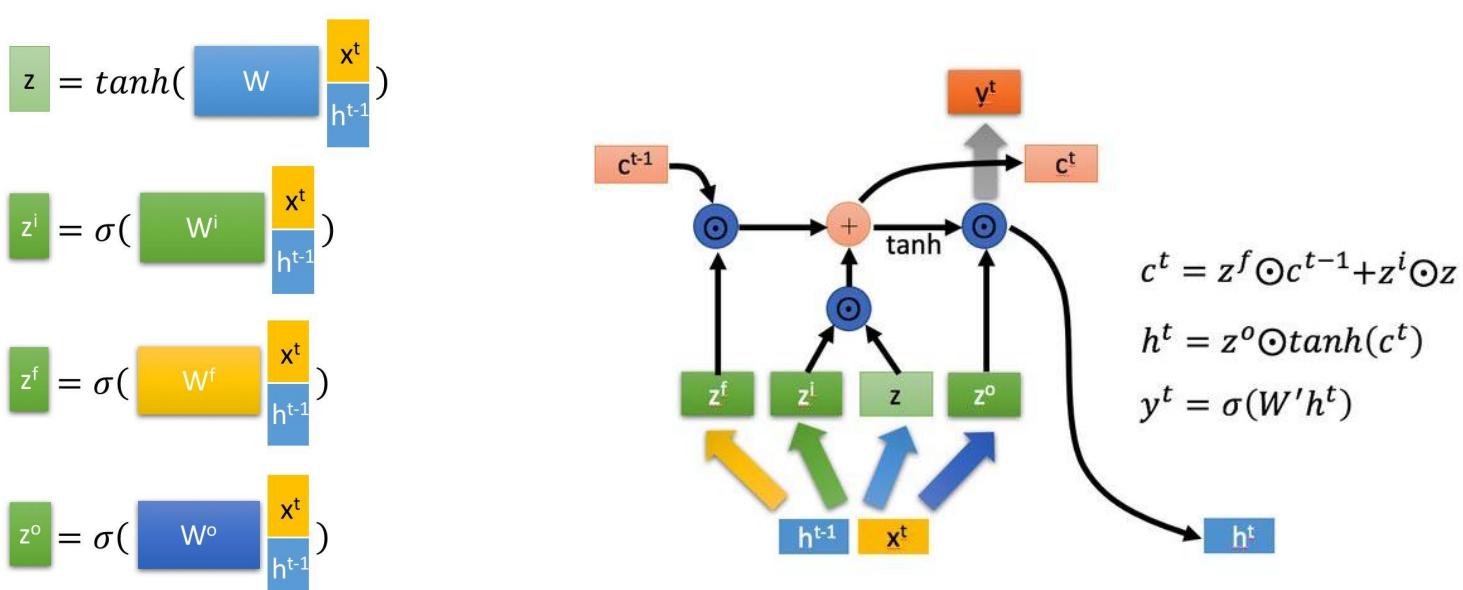
LSTM 结构 (图右) 和普通RNN的主要输入输出区别如下所示:



相比RNN只有一个传递状态 h^t ，LSTM有两个传递状态，一个 c^t (cell state, 细胞态，长期记忆)，和一个 h^t (hidden state, 隐态，记忆体，短期记忆)。（Tips: RNN中的 h^t 对于LSTM中的 c^t ）。其中对于传递下去的 c^t 改变得很慢，通常输出的 c^t 是上一个状态传过来的 c^{t-1} 加上一些数值，而 h^t 则在不同节点下往往会有很大的区别。

下面具体对 LSTM 的内部结构来进行剖析：

首先使用 LSTM 的当前输入 x^t 和上一个状态传递下来的 h^{t-1} 拼接训练得到四个状态。



其中， z^f ， z^i ， z^o 是由拼接向量乘以权重矩阵之后，再通过一个 *sigmoid* 激活函数转换成0到1之间的数值，来作为一种门控状态，分别表示：

z^i 表示输入门， z^f 表示遗忘门， z^o 表示输出门。

而 z 则是将结果通过一个 *tanh* 激活函数将转换成-1到1之间的值（这里使用 *tanh* 是因为这里是将其做为输入数据，而不是门控信号）， z 表示候选态，由上一时刻的短期记忆 h^{t-1} 和当前时刻的输入特征 x^t 决定。

c^t (cell state, 细胞态, 长期记忆)， h^t (hidden state, 记忆体, 短期记忆)

⊗ 是Hadamard Product，也就是操作矩阵中对应的元素相乘，因此要求两个相乘矩阵是同型的。⊕ 则代表进行矩阵加法。

LSTM内部主要有三个阶段：

1. **遗忘门**。这个阶段主要是对上一个节点传进来的输入进行**选择性**遗忘。简单来说就是会“忘记不重要的，记住重要的”。

具体来说是通过计算得到的 z^f (f 表示forget) 来作为忘记门控，来控制上一个状态的 c^{t-1} 哪些需要留哪些需要忘。

2. **输入门**。这个阶段将这个阶段的输入有选择性地进行“记忆”。主要是会对输入 x^t 进行选择记忆。哪些重要则着重记录下来，哪些不重要，则少记一些。当前的输入内容由前面计算得到的 z 表示。而选择的门控信号则是由 z^i (i 代表input) 来进行控制。

将上面两步得到的结果相加，即可得到传输给下一个状态的 c^t 。也就是上上图中的第一个公式。

3. **输出门**。这个阶段将决定哪些将会被当成当前状态的输出。主要是通过 z^o 来进行控制的。并且还对上一阶段得到的 c^o 进行了放缩（通过一个*tanh*激活函数进行变化）。

与普通RNN类似，输出 y^t 往往最终也是通过 h^t 变化得到。

以上，就是LSTM的内部结构。通过门控状态来控制传输状态，记住需要长时间记忆的，忘记不重要的信息；而不像普通的RNN那样只能够“呆萌”地仅有一种记忆叠加方式。对很多需要“长期记忆”的任务来说，尤其好用。**但也因为引入了很多内容，导致参数变多，也使得训练难度加大了很多。**因此很多时候我们往往会使用效果和LSTM相当但参数更少的GRU来构建大训练量的模型。

例子：

```
1 # 测试集变array并reshape为符合RNN输入要求：[送入样本数, 循环核时间展开步数, 每个时间步输入特征个数]
2 x_test, y_test = np.array(x_test), np.array(y_test)
3 x_test = np.reshape(x_test, (x_test.shape[0], 60, 1))
4
5 model = tf.keras.Sequential([
6     LSTM(80, return_sequences=True), # 每个时间步都输出h_t
7     Dropout(0.2),
8     LSTM(100), # 只在最后时间步输出h_t
9     Dropout(0.2),
10    Dense(1)
11])
```

GRU

参考文献:

1. GRU是什么? RNN、LSTM分别是什么? : https://blog.csdn.net/Frank_LJiang/article/details/103142557

GRU

GRU的输入输出结构与普通的RNN是一样的, **它的提出是为了解决LSTM计算过于复杂的问题**。有一个当前的输入 x^t , 和上一个节点传递下来的隐状态(hidden state) h^{t-1} , 这个隐状态包含了之前节点的相关信息。

结合 x^t 和 h^{t-1} , GRU会得到当前隐藏节点的输出 y^t 和传递给下一个节点的隐状态 h^t 。



那么, GRU到底有什么特别之处呢? 下面来对它的内部结构进行分析!

首先, 我们先通过上一个传输下来的状态 h^{t-1} 和当前节点的输入 x^t 来获取两个门控状态。如下图所示, 其中 r 控制重置的门控(reset gate), z 为控制更新的门控(update gate)。

Tips: σ 为sigmoid函数, 通过这个函数可以将数据变换为0-1范围内的数值, 从而来充当门控信号。

与LSTM分明的层次结构不同, 得到门控信号之后, 首先使用重置门控来得到“重置”之后的数据。

$$h^{t-1'} = h^{t-1} \odot r$$

再将 $h^{t-1'}$ 与输入 x^t 进行拼接, 再通过一个tanh激活函数来将数据放缩到-1~1的范围内。即得到如下图所示的 h' 。

$$h' = \tanh(W \begin{bmatrix} x^t \\ h^{t-1'} \end{bmatrix})$$

这里的 h' 主要是包含了当前输入的 x^t 数据。有针对性地对 h' 添加到当前的隐藏状态。

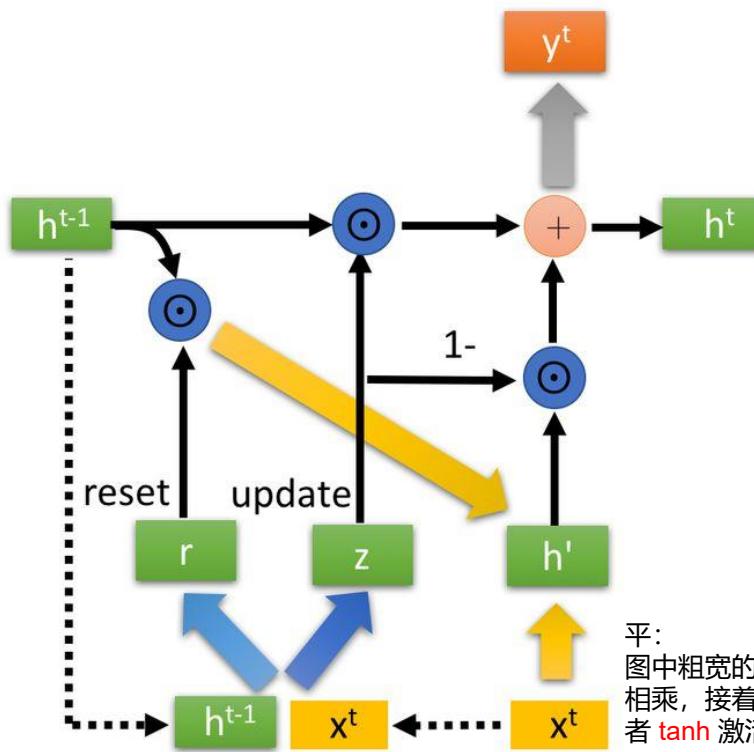
$$r = \sigma(W^r \begin{pmatrix} x^t \\ h^{t-1} \end{pmatrix})$$

$$z = \sigma(W^z \begin{pmatrix} x^t \\ h^{t-1} \end{pmatrix})$$

$$h^{t-1'} = h^{t-1} \odot r$$

$$h' = \tanh(W \begin{pmatrix} x^t \\ h^{t-1'} \end{pmatrix})$$

$$h^t = z \odot h^{t-1} + (1 - z) \odot h'$$



平：
图中粗宽的箭头表示有系数矩阵 W 相乘，接着经过激活函数 sigmoid 或者 \tanh 激活。即表示 NN layer。

◎ 是Hadamard Product，也就是操作矩阵中对应的元素相乘，因此要求两个相乘矩阵是同型的。⊕ 则代表进行矩阵加法操作。

最后介绍GRU最关键的一个步骤，我们可以称之为”更新记忆“阶段。在这个阶段，我们同时进行了遗忘和记忆两个步骤。我们使用了先前得到的更新门控 z (update gate)。 z 相当于遗忘门， $1-z$ 相当于输入门

更新表达式：

$$h^t = z \odot h^{t-1} + (1 - z) \odot h'$$

首先再次强调一下，门控信号（这里的 z ）的范围为0~1。门控信号越接近1，代表“记忆”下来的数据越多；而越接近0则代表“遗忘”的越多。

GRU很聪明的一点就在于，使用了同一个门控 z ，即同时可以进行遗忘和记忆（LSTM则要使用多个门控）。

- $z \odot h^{t-1}$ ：表示对原本隐藏状态的选择性“遗忘”。这里的 z 可以想象成遗忘门 (forget gate)，忘记 h^{t-1} 维度中一些不重要的信息。
- $(1 - z) \odot h'$ ：表示对包含当前节点信息的 h' 进行选择性“记忆”。与上面类似，这里的 $(1 - z)$ 同理会记住 h' 中重要的，忘记 h' 维度中的一些不重要的信息。或者，这里我们更应当看做是对 h' 维度中的某些信息进行选择。
- $h^t = z \odot h^{t-1} + (1 - z) \odot h'$ ：结合上述，这一步的操作就是忘记传递下来的 h^{t-1} 中的某些维度信息，并加入当前节点输入的某些维度信息。

可以看到，这里的遗忘 z 和选择 $(1 - z)$ 是联动的。也就是说，对于传递进来的维度信息，我们会进行选择性遗忘，则遗忘了多少权重 (z)，我们就会使用包含当前输入的 h' 中所对应的权重进行弥补 $(1 - z)$ 。以保持一种“恒定”状态。

```
1 # 测试集变array并reshape为符合RNN输入要求: [送入样本数, 循环核时间展开步数, 每个时间步输入特征个数]
2 x_test, y_test = np.array(x_test), np.array(y_test)
3 x_test = np.reshape(x_test, (x_test.shape[0], 60, 1))
4
5 model = tf.keras.Sequential([
6     GRU(80, return_sequences=True),
7     Dropout(0.2),
8     GRU(100),
9     Dropout(0.2),
10    Dense(1)
11])
```

LSTM与GRU的关系

GRU是在2014年提出来的，而LSTM是1997年。他们的提出都是为了解决相似的问题，那么GRU难免会参考LSTM的内部结构。那么他们之间的关系大概是怎么样的呢？这里简单介绍一下。大家看到 r (reset gate)实际上与他的名字有点不符。我们仅仅使用它来获得了 h' 。那么这里的 h' 实际上可以看成对应于LSTM中的hidden state；上一个节点传下来的 h^{t-1} 则对应于LSTM中的cell state。 z 对应的则是LSTM中的 z^f forget gate，那么 $(1 - z)$ 我们似乎就可以看成是选择门 z^i 了。

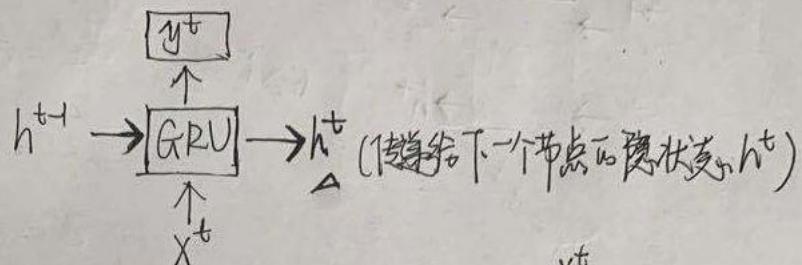
总结

GRU输入输出的结构与普通的RNN相似，其中的内部思想与LSTM相似。与LSTM相比，GRU内部少了一个“门控”，参数比LSTM少，但是却也能够达到与LSTM相当的功能。考虑到硬件的**计算能力和时间成本**，因而很多时候我们也就选择更加“实用”的GRU啦。

参考文献：

- GRU原理理解：https://blog.csdn.net/qq_38147421/article/details/107694477
- (与上一篇引用文章：GRU是什么？RNN、LSTM分别是什么？：https://blog.csdn.net/Frank_LJiang/article/details/103142557 基本相同，最后的下面这部分，是主要的不同之处。)

GRU：改变了RNN的隐藏层，改善梯度消失问题。
其结构和普通的RNN是一样的。



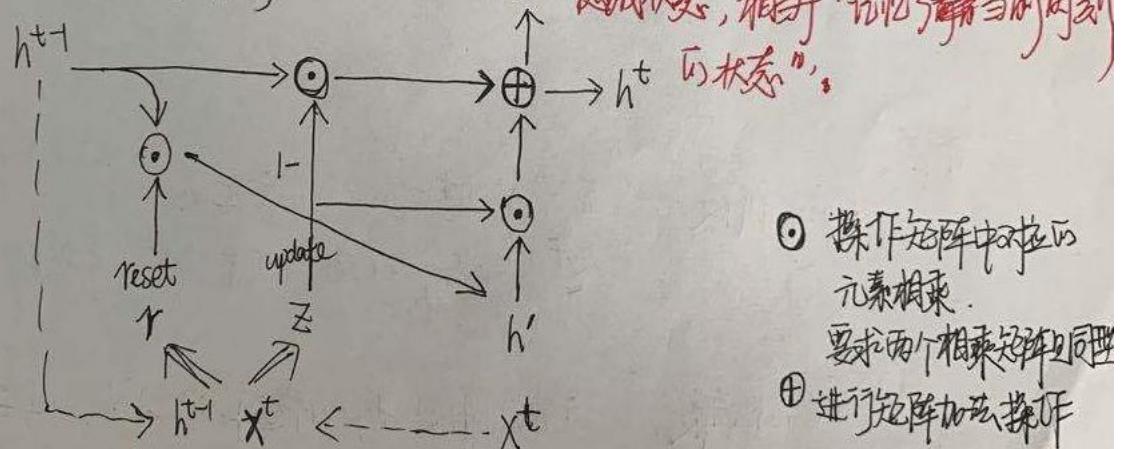
γ：控制重置门控。reset gate. $r = \sigma(W_r x^t + h^{t-1})$

ζ：控制更新门控。update gate $\zeta = \sigma(W_\zeta x^t + h^{t-1})$

σ 为 sigmoid 函数

将数值映射到 0-1 之间

得到门控信号后，首先使用重置门控，得到重置后的数据 $h^{t-1}' = h^{t-1} \odot r$
再将 h^{t-1}' 与输入 x^t 拼接，再通过激活函数 \tanh 有数据从范围 (-1, 1) 之间。
得到 $h' = \tanh(W_h x^t + h^{t-1}')$ h' 主要包含了 x^t 的数据。有针对性地对 h' 添加到当前的
隐藏状态，相当于“记忆了当前时刻”。



- ① 操作矩阵中对应的元素相乘。
要求两个相乘矩阵同理
- ② 进行矩阵加法操作

* GRU 最 important 的一个步骤，“更新记忆”阶段。

更新表达式： $h^t = (1 - \zeta) \odot h^{t-1} + \zeta \odot h'$ 遗忘+记忆区 (0,1)

遗忘门 忘记 h^{t-1} 中一些重要的信息。

对原本隐藏状态的选择性“遗忘”

$\zeta \rightarrow 1$ ：代表“记忆”的数据越多

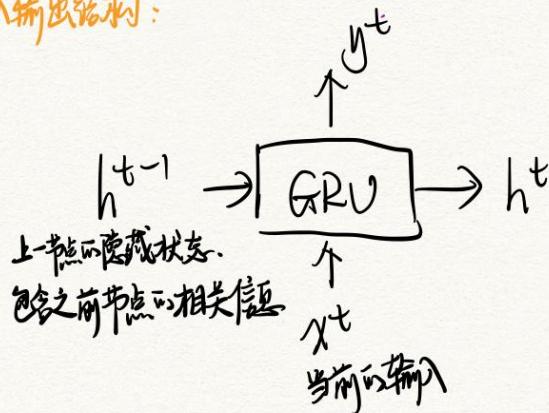
$\zeta \rightarrow 0$ ：代表“遗忘”的数据越多。

面试补充

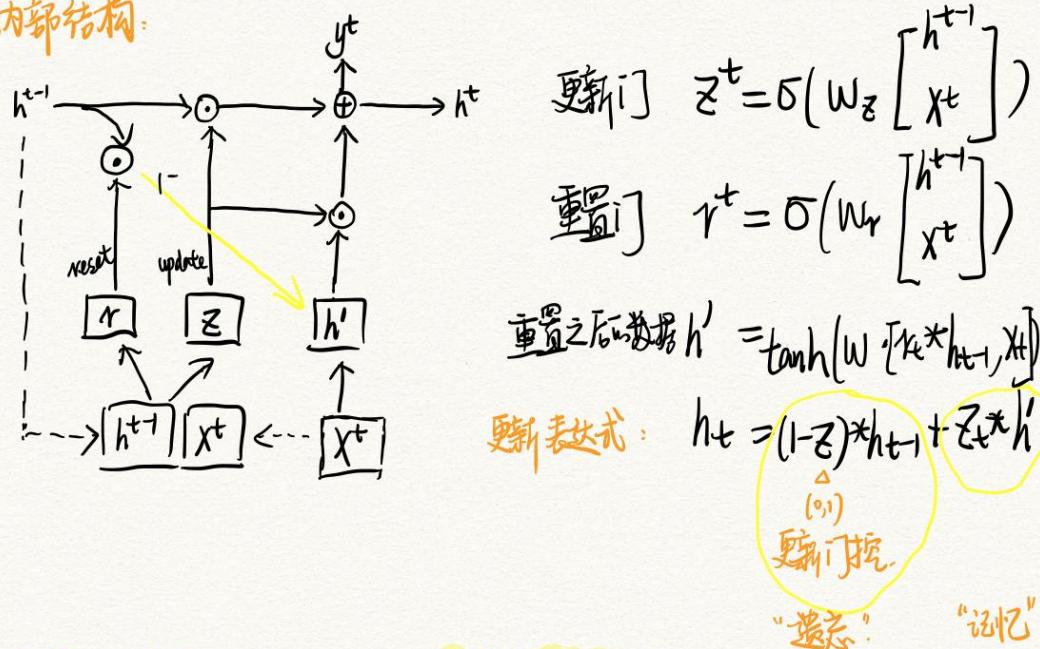
画出GRU的结构：

GRU只有两个门。GRU将LSTM中的输入门和遗忘门合二为一，称为更新门（update gate），控制前边记忆信息能够继续保留到当前时刻的数据量；另一个门称为重置门（reset gate），控制要遗忘多少过去的信息。

GRU的输入输出结构：

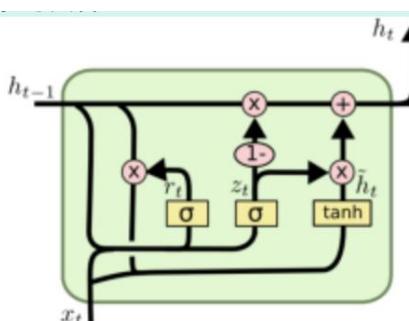


GRU的内部结构：



GRU最聪明：用一个门搞了就可以同时进行遗忘和选择记忆（LSTM则需要多门搞）

<https://zhidao.baidu.com/question/38147421.html>



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

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

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

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

GRU有两个门：更新门，输出门

解析：如果不会画GRU，可以画LSTM或者RNN。再或者可以讲解GRU与其他两个网络的联系和区别。不要直接就说不会。

https://www.csdn.net/nav/q_38