

时间序列分析 (4) RNN/LSTM



随风

大数据、人工智能

关注他

81 人赞同了该文章

1 前言

在上一章中，我们介绍了线性回归，从特征的角度建立了多元线性模型。本章将介绍两个深度学习模型（非线性模型），RNN/LSTM。时间序列分析 (3) Linear Regression:

随风：时间序列分析 (3) Linear Regression

zhuanlan.zhihu.com



语言：python3

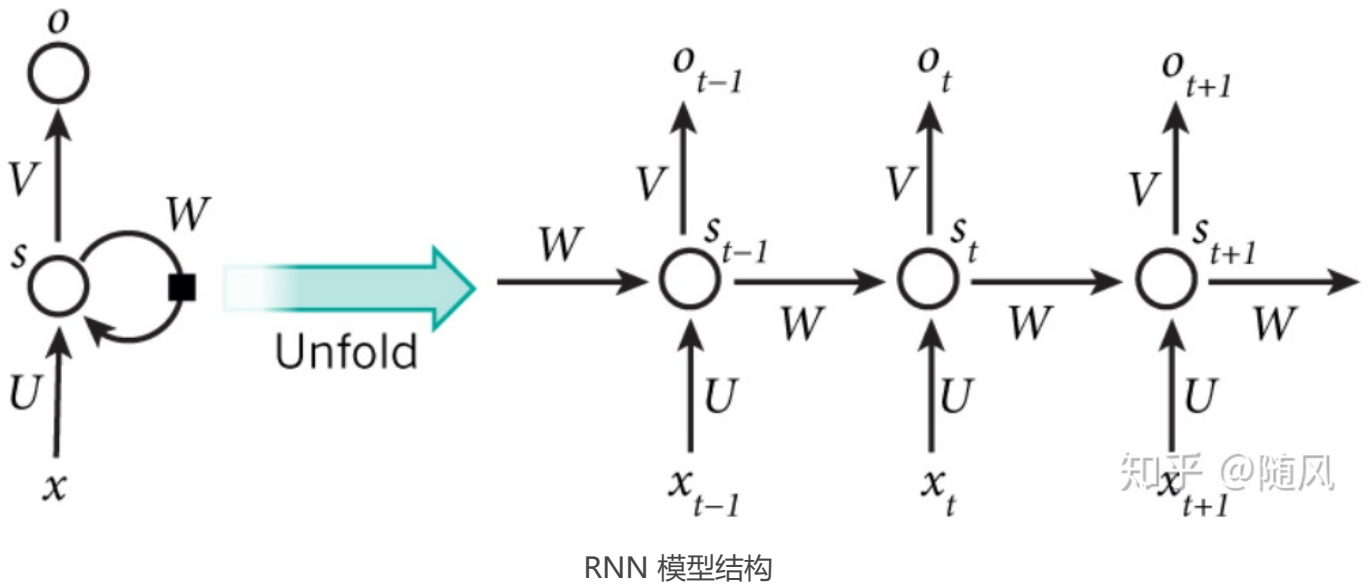
数据集：余额宝在2014-03-01~2014-08-31期间每日申购的总金额（数据来自天池大赛）

数据下载地址：tianchi.aliyun.com/comp

2 RNN (Recurrent Neural Network) 循环神经网络

2.1 RNN 概述

传统神经网络 (DNN) 无法对时间序列建模，上一层神经元的输出只能传递给下一层神经元。而在循环神经网络 (RNN) 中，神经元的输出在下一时刻是可以传递给自身的，同时还输出一个隐藏层状态，给当前层在处理下一个样本时使用，**它可以看作是带自循环反馈的全连接神经网络**。很多任务的时序信息很重要，即一个样本中前后输入的信息是有关联的。样本出现时间顺序信息对语音识别、自然语言处理、视频识别等问题很重要，所以对于这类问题，可以使用 RNN 建模，经典的 RNN 模型结构如图：



上图中左边是没有按时序展开的图（与DNN相似），右边是按时序展开的图，我们重点讨论右边的图。

x_t 表示 t 时刻的输入， s_t 表示 t 时刻模型的隐藏层状态， o_t 表示 t 时刻的输出，其中 x_t 、 s_t 、 o_t 均为向量。 U 、 W 、 V 这三个矩阵是模型的线性映射参数，它们在整个网络中是共享的，可以在时间上共享不同位置的统计强度，这正是体现了 RNN 模型的“循环反馈”的思想。当序列数据中的某些部分在多个位置出现时，这种参数共享机制就显得尤为重要了。例如，识别两个单词 what、how 中的“w”字母，我们希望模型通过参数共享机制可以学习到字母“w”的抽象特征，从而无论这个字母出现在什么位置，模型都能够识别它。

2.2 激活函数

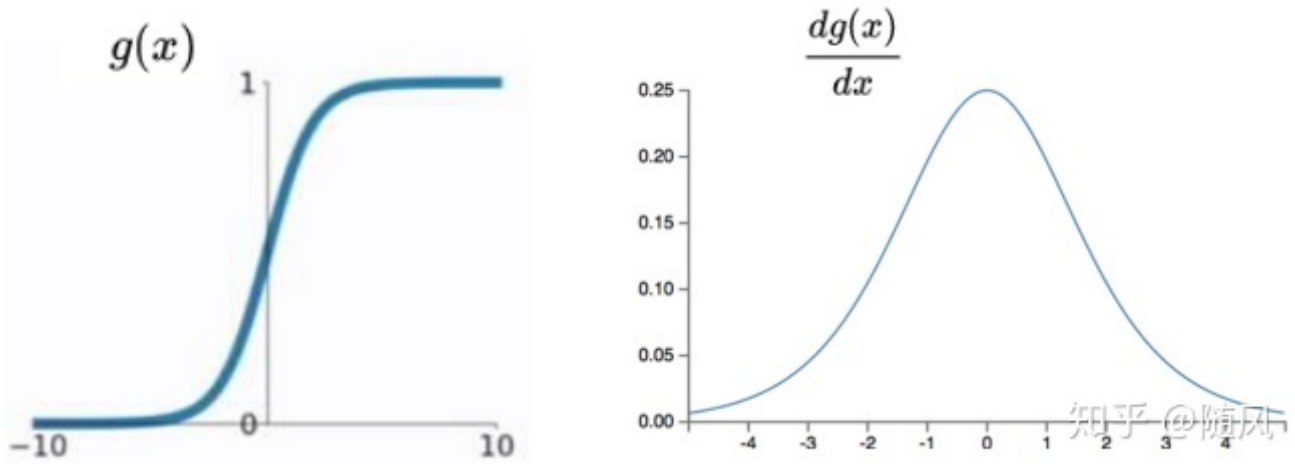
从生物学的角度，对于某一个神经元，并不是在给定激励信号后一定会有输出信号，神经元会选择性地输出，比如对于一个很小的激励，神经元可能不会产生输出信号。在人工神经网络中，为了模拟神经元的这一特性，人们引入了激活函数。

此外，如果不使用激活函数，每一层输出都是上层输入的线性函数，无论神经网络有多少层，输出都是输入的线性组合。激活函数给神经元引入了非线性因素，使得神经网络可以逼近任意非线性函数，这样神经网络就可以应用到众多的非线性模型中。

常见的激活函数有 sigmoid、tanh、relu、softmax：

(1) sigmoid 函数和一阶导数如下：

$$g(x) = \frac{1}{1 + e^{-x}}, \quad \frac{dg(x)}{dx} = g(x)(1 - g(x))$$

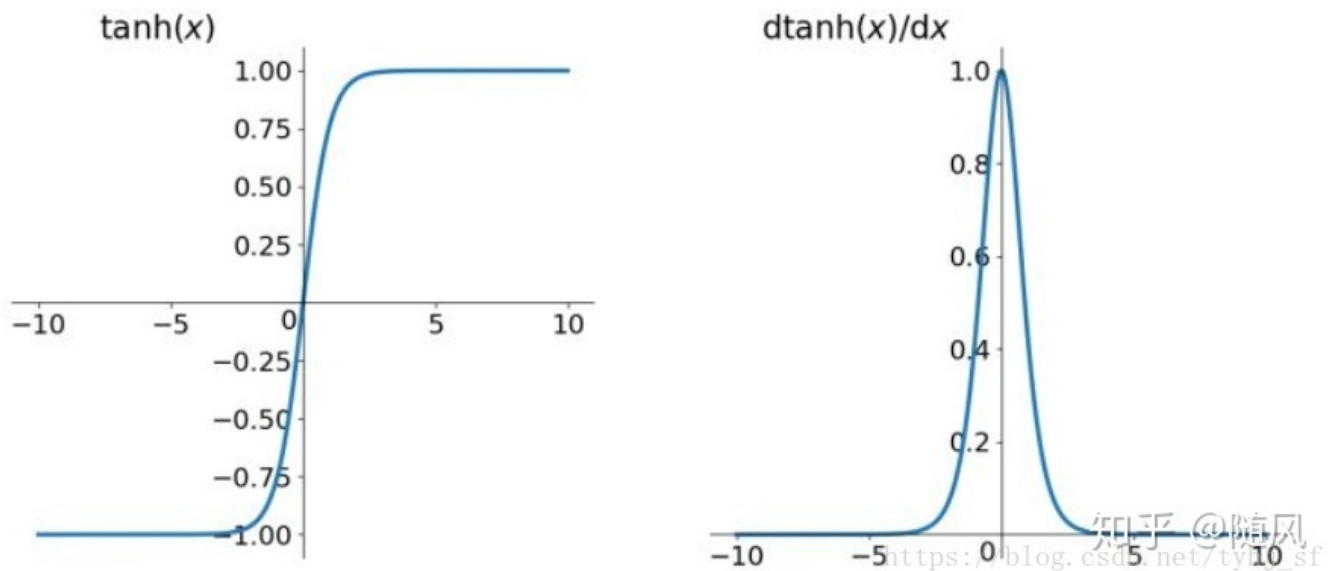


sigmoid函数（左），一阶导数（右）

sigmoid 函数是将取值为 $(-\infty, +\infty)$ 的数据映射到 $(0, 1)$ 之间，取 $-\infty$ 的时候映射为 0，取 $+\infty$ 的时候映射为 1，取 0 的时候映射为 0.5。它的一阶导数取值范围为 $0 \sim 0.25$ ，数据越趋近 0，导数越趋于 0.25，数据越趋近 ∞ ，导数越趋于 0。

(2) tanh 函数和一阶导数如下：

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \quad \frac{d\tanh(x)}{dx} = 1 - \tanh^2(x)$$

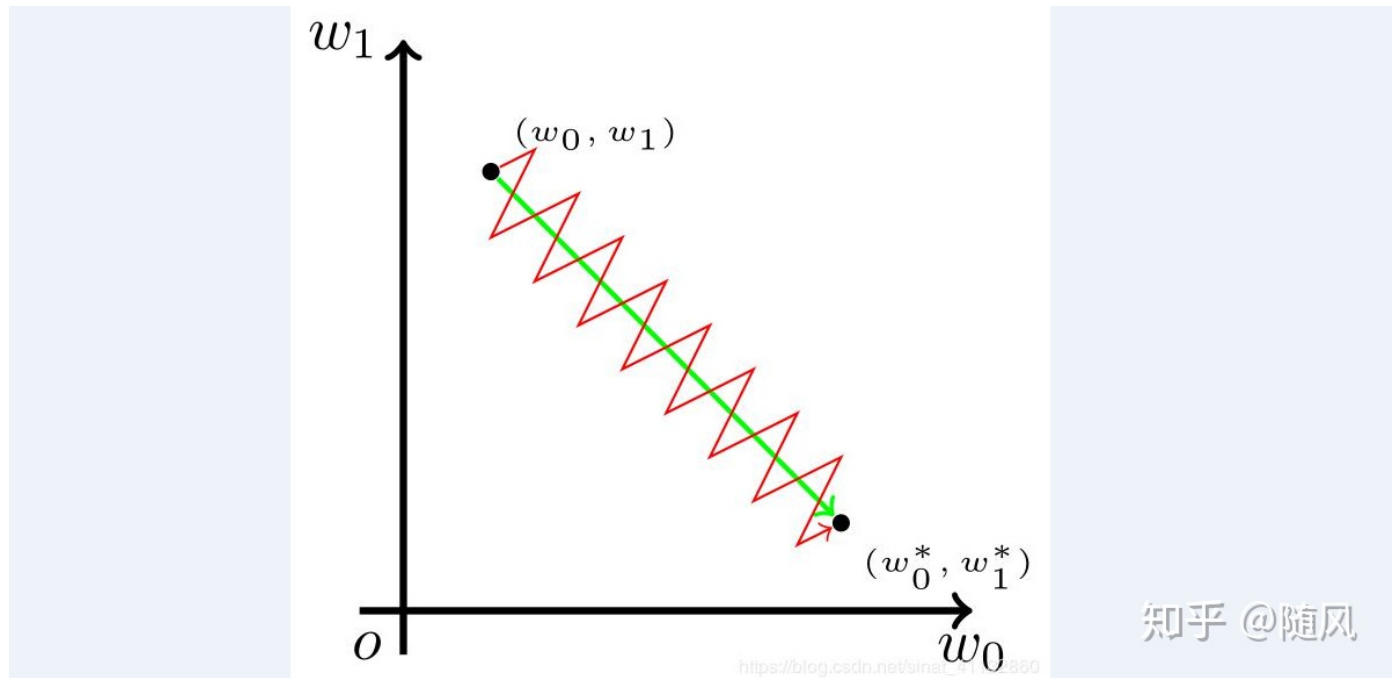


tanh函数（左），一阶导数（右）

tanh 函数是将取值为 $(-\infty, +\infty)$ 的数据映射到 $(-1, 1)$ 之间，取 $-\infty$ 的时候映射为 -1，取 $+\infty$ 的时候映射为 1，取 0 的时候映射为 0。它的一阶导数取值范围为 $0 \sim 1$ ，数据越趋近 0，导数越趋于 1，数据越趋近 ∞ ，导数越趋于 0。相比于 sigmoid 函数，tanh 函数是关于 0 中心对称的。

下图以二维空间为例说明 0 中心对称问题，其中 (w_0, w_1) 为模型的初始参数， (w_0^*, w_1^*) 为模型最优解。模型走绿色箭头能够最快收敛，如果采用 sigmoid 作为激活函数，则后面的每一层

输入均为正，此时模型为了收敛，可能走类似红色折线箭头逼近最优解，收敛速度就会慢很多。

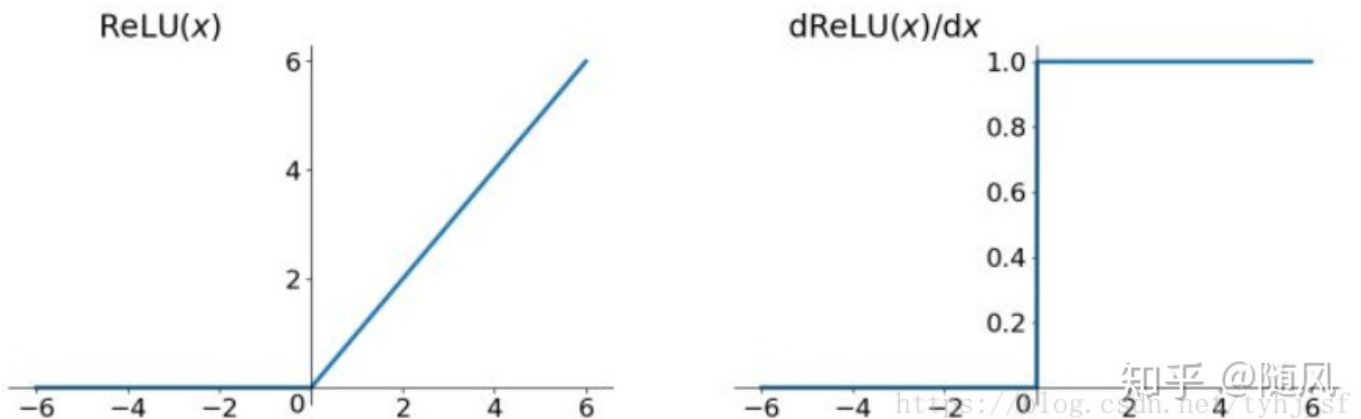


模型参数训练过程

(3) 通过上面分析，sigmoid 函数的一阶导数范围为 $0 \sim 0.25$ ，tanh 函数的一阶导数范围为 $0 \sim 1$ ，如果输入数据稍微大一点，梯度就会变得非常小，这样在链式求导法则的累乘过程中，整体梯度非常容易趋近于 0，发生梯度消失问题。Relu 函数可以解决深度神经网络中梯度消失的问题。Relu 函数和一阶导数如下：

$$\text{relu}(x) = \max(0, x)$$

$$\frac{d\text{relu}(x)}{dx} = 1, \quad x > 0; \quad \frac{d\text{relu}(x)}{dx} = 0, \quad x < 0$$



relu函数 (左) , 一阶导数 (右)

Relu 函数是一种分段函数（注：Relu 函数本身是非线性函数，但是两个分段部分都是线性函数）：它弥补了 sigmoid 函数以及 tanh 函数的**梯度消失问题**（当输入为正，梯度恒为 1；当输入为负，梯度恒为 0）。此外，相比于 sigmoid 函数或 tanh 函数，它的计算只有线性关系，因此运

算速度会快很多。但是 Rule 函数也有问题，首先它在 0 点处是不可导的，通常我们可以采用中间导数代替 (0.5)，其次当输入为负的时候，神经元不会被激活，造成梯度消失问题。关于 Relu 函数的改进版本也有很多，比如 Leaky Relu，但是 Relu 依然是深度学习中使用的最广泛的激活函数。

2.3 RNN 前向传播法

令 y_t 表示 t 时刻的真实值， L_t 表示 t 时刻的损失函数， \tilde{y}_t 表示 t 时刻模型的预测值。则 t 时刻隐藏层状态 s_t 为：

$$s_t = \phi(Ux_t + Ws_{t-1} + b), \text{ 其中 } \phi \text{ 为激活函数, 一般选择 } \tanh, b \text{ 为偏置项。}$$

则 t 时刻输出 o_t 为：

$$o_t = Vs_t + c, \text{ 其中 } c \text{ 为偏置项。}$$

最终模型在 t 时刻的预测值 \tilde{y}_t 为：

$$\tilde{y}_t = \sigma(o_t), \text{ 其中 } \sigma \text{ 为激活函数, 通常选择 softmax 函数。}$$

2.4 RNN 反向传播法 (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 的核心思想和 BP 相同，沿着需要优化的参数的负梯度方向不断寻找最优的点。当然这里的 BPTT 和 DNN 中的 BP 也有很大的不同，这里所有的 U、W、V 在序列的各个位置是共享的，反向传播时我们更新的是相同的参数。由于序列在每个时刻都有损失函数，因此最终的损失 L 为：

$$L = \sum_{t=1}^n L_t, \text{ 因此可以得到 U、W、V 的偏导数, 先来看 V 的偏导数:}$$

$$\frac{\partial L}{\partial V} = \sum_{t=1}^n \frac{\partial L_t}{\partial \tilde{y}_t} \frac{\partial \tilde{y}_t}{\partial o_t} \frac{\partial o_t}{\partial V}, \text{ U、W 比较复杂, 下面以 } t=3 \text{ 时刻为例。}$$

$$\frac{\partial L_3}{\partial W} = \frac{\partial L_3}{\partial \tilde{y}_3} \frac{\partial \tilde{y}_3}{\partial o_3} \frac{\partial o_3}{\partial s_3} \frac{\partial s_3}{\partial W} + \frac{\partial L_3}{\partial \tilde{y}_3} \frac{\partial \tilde{y}_3}{\partial o_3} \frac{\partial o_3}{\partial s_3} \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial W} + \frac{\partial L_3}{\partial \tilde{y}_3} \frac{\partial \tilde{y}_3}{\partial o_3} \frac{\partial o_3}{\partial s_3} \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial W}$$

$$\frac{\partial L_3}{\partial U} = \frac{\partial L_3}{\partial \tilde{y}_3} \frac{\partial \tilde{y}_3}{\partial o_3} \frac{\partial o_3}{\partial s_3} \frac{\partial s_3}{\partial U} + \frac{\partial L_3}{\partial \tilde{y}_3} \frac{\partial \tilde{y}_3}{\partial o_3} \frac{\partial o_3}{\partial s_3} \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial U} + \frac{\partial L_3}{\partial \tilde{y}_3} \frac{\partial \tilde{y}_3}{\partial o_3} \frac{\partial o_3}{\partial s_3} \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial U}$$

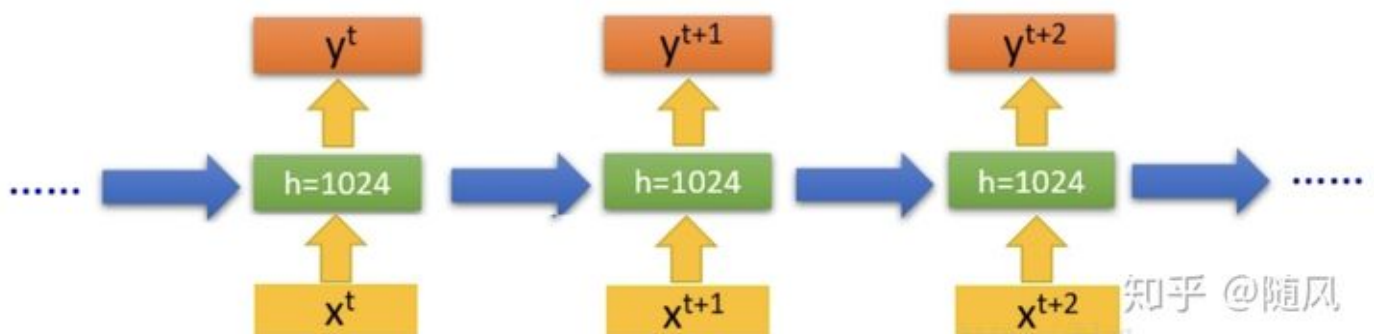
对于 s_t ，如果激活函数选择 sigmoid 或 tanh，它们的一阶导数在 0~1 之间，随着时间序列的不断深入，小数的累乘就会导致梯度越来越小直到接近于 0，这就会引起梯度消失现象。**这里需要指出，由于 tanh 的一阶导数范围更大，且关于 0 中心对称，因此收敛速度要快于 sigmoid，而梯度消失的速度要慢于 sigmoid。**如果激活函数选择 Relu，当输入为正时，梯度恒为 1，因此不会出现累乘后梯度消失的问题，但是在累加的过程中，梯度会越来越大，容易引起梯度爆炸的问题；当输入为负时，梯度恒为 0，会产生梯度消失问题。**目前，关于梯度问题并不是仅通过激活函数来解决，通常结合参数初始化、批正则等方法共同解决。**

利用 BPTT 算法训练网络时容易出现梯度消失的问题，当序列很长的时候问题尤其严重，因此 RNN 模型一般不能直接应用。而较为广泛使用的是 RNN 的一个特例 LSTM。

2.5 RNN 的分类

(1) 单隐层 RNN

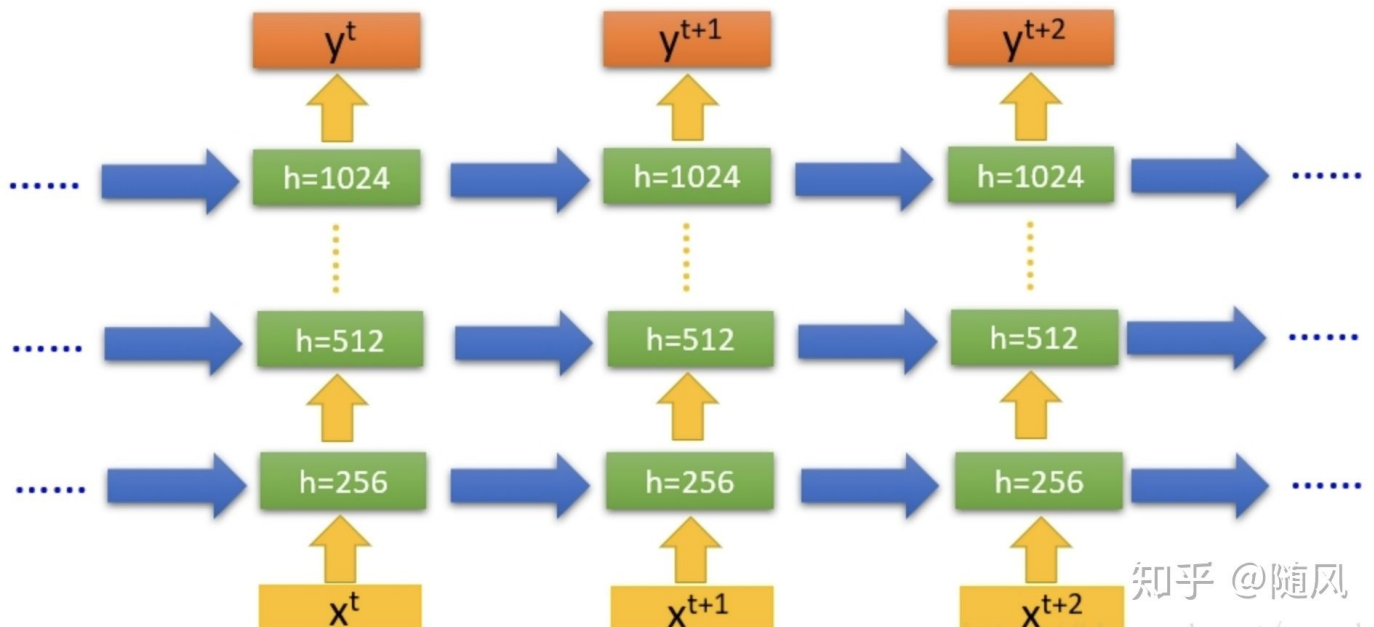
从输入到输出，只有一层隐藏层，这是最基本的 RNN 模型，上面对于 RNN 的讲解，也是基于单隐层模型。



单隐层 RNN 模型

(2) 多隐层 RNN

单隐层 RNN 可以看作既“深”又“浅”的网络。如果我们把 RNN 网络按时间展开，它的时间链路很长，因此可以看作是一个非常深的网络。另一方面，如果只看同一时刻输入到输出（只包含一个隐藏层），网络是非常浅的。**增加循环神经网络的深度，主要是增加隐藏层的数量。**

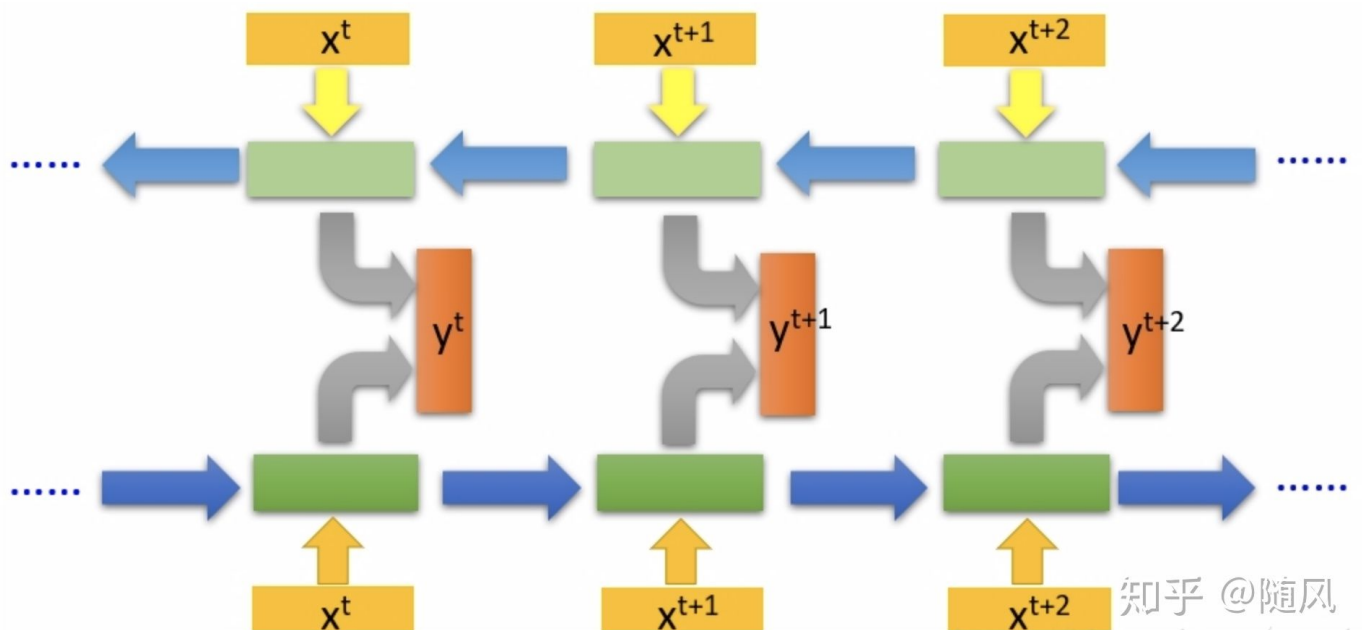


多隐层 RNN 模型

(3) 双向 RNN

在一些任务中，一个时刻的输出不但和序列前面的信息有关，也和序列后面的信息有关。比如给定一个句子，其中一个词的词性由它的上下文决定，即：包含左右两边的信息。因此，在这些任务中，我们可以增加一个按照时间的逆序来传递信息的网络层，来增强网络的能力。

双向循环神经网络 (bidirectional recurrent neural network, Bi-RNN) 由两层循环神经网络组成，它们的输入相同，只是信息传递的方向不同。



双向 RNN 模型

3 LSTM (Long Short-Term Memory) 长短期记忆网络

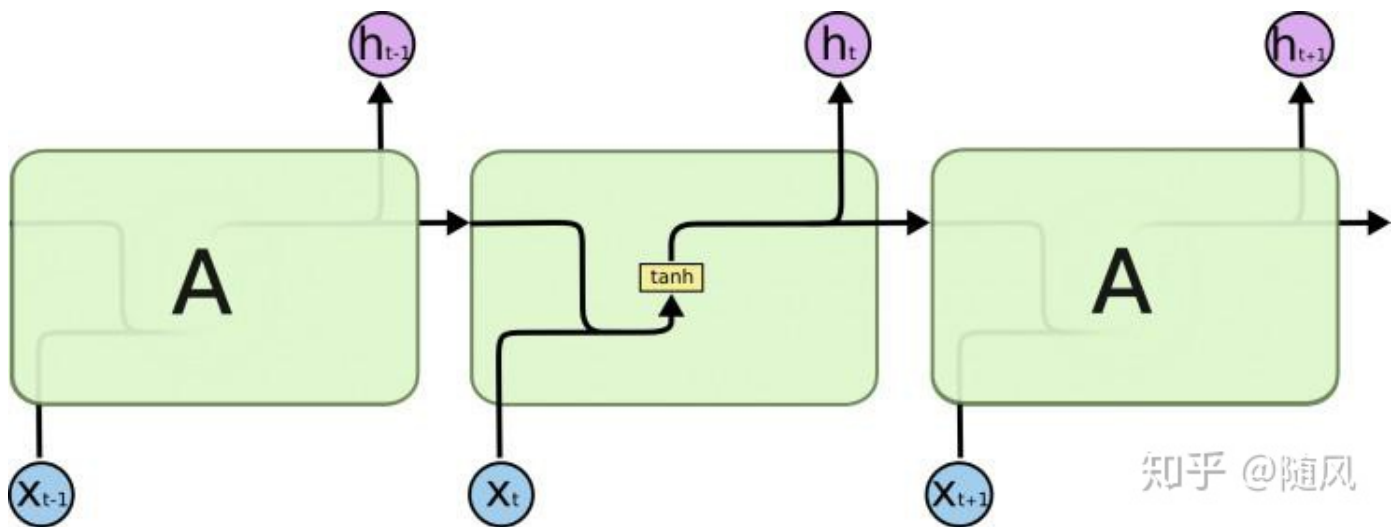
3.1 LSTM 计算模型

LSTM 是一种改进之后的循环神经网络，可以解决 RNN 无法处理长距离的依赖的问题。针对 RNN 存在的问题，LSTM 主要有两点改进：

设置专门的变量 C_t 来存储单元状态 (Cell State)，从而使网络具有长期记忆。

引入“门运算”，将梯度中的累乘变为累加，解决梯度消失问题。

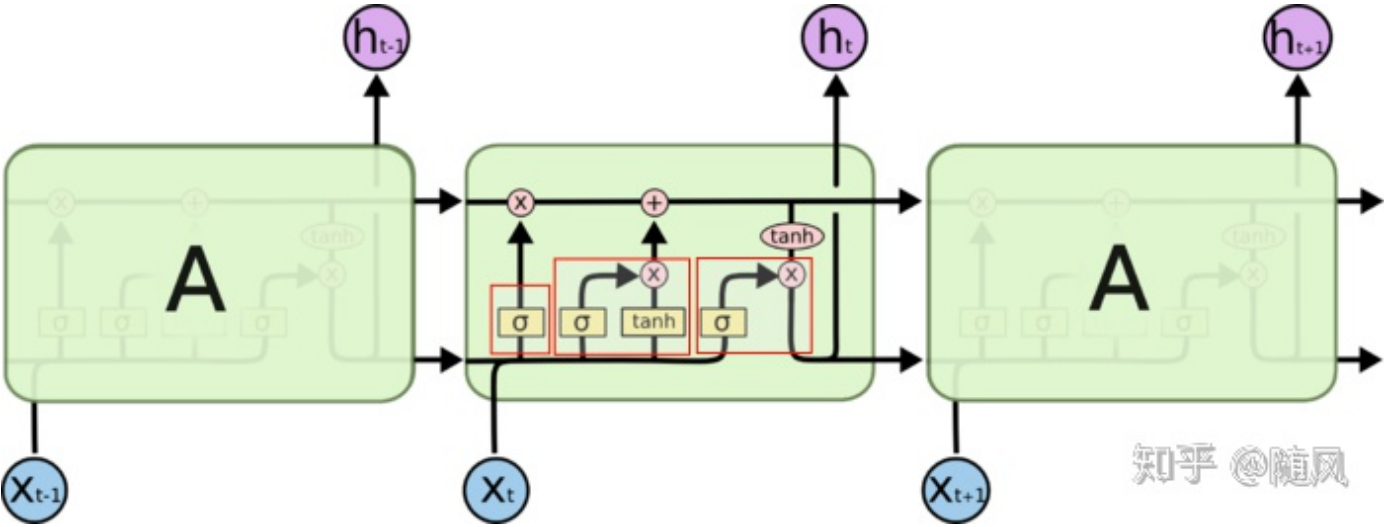
在 RNN 模型中，输入为上一时刻隐层状态 h_{t-1} ，当前时刻输入值 x_t ，输出为当前时刻隐层状态 h_t ，中间的运算只包含一个激活函数。



RNN模型

在 LSTM 模型中，输入为上一时刻的隐层状态 h_{t-1} ，上一时刻的单元状态 C_{t-1} ，当前时刻输入值 x_t ，输出为当前时刻隐层状态 h_t ，当前时刻的单元状态 C_t 。相比于 RNN 内部的计算要复杂。通常称内部为三个门运算：遗忘门 (forget)、输入门 (input)、输出门 (output)，三个门分别用红线框出。

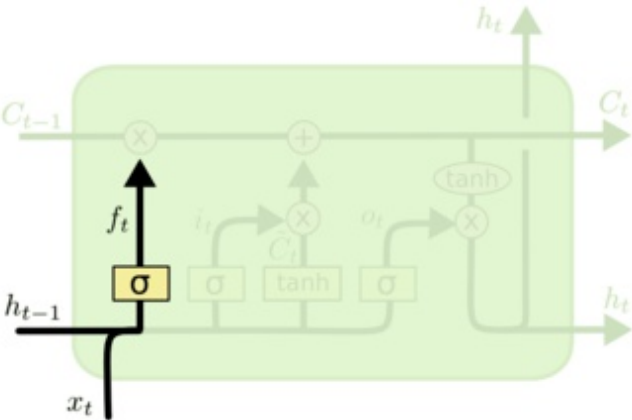
这里解释一下单元状态，以 t 时刻为例，我们把图中间的绿色框叫做 t 时刻的单元，包括了输入输出以及内部的运算，它的状态用向量 C_t 表示。左右两边的绿色框分别表示 $t-1$ ， $t+1$ 时刻的单元。**注意，它们是同一个单元在不同时刻的实例，并不是多个单元。**



LSTM模型

(1) 遗忘门

遗忘门用来计算哪些信息需要忘记，通过 sigmoid 处理后为 0 到 1 的值，1 表示全部保留，0 表示全部忘记： σ 为 sigmoid 函数。注意这里 f_t 、 C_{t-1} 都是向量。



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

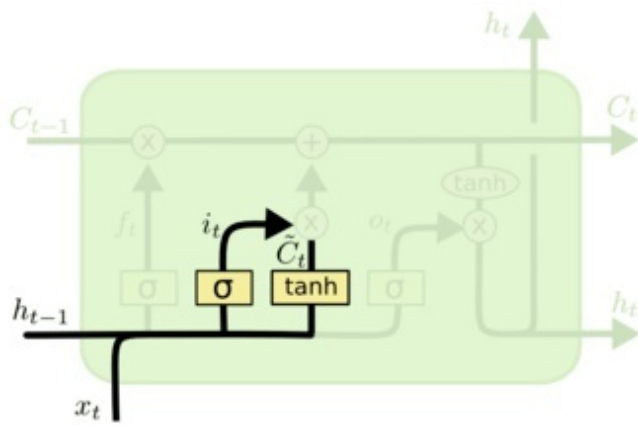
遗忘门 (forget)

(2) 输入门

输入门用来计算哪些当前信息需要保存，分为两部分：

第一部分： i_t 的取值范围是 0 到 1，1 表示全部保留，0 表示全部丢弃。

第二部分： \tilde{C}_t 表示新信息，结合这两部分来创建一个新记忆。



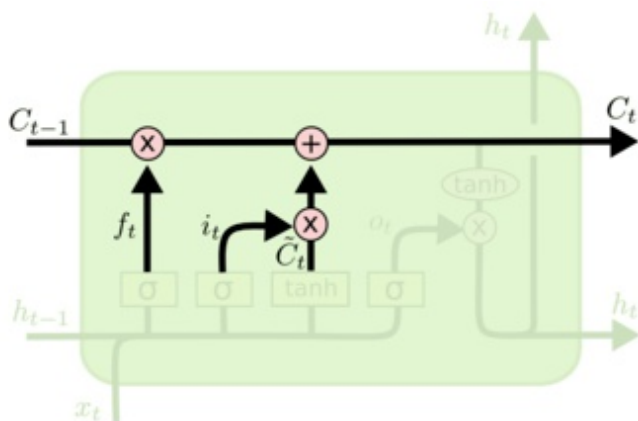
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

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

知乎 @随风

输入门 (input)

结合遗忘门和输入门，可以计算当前单元状态 C_t 。从公式来看， C_t 表示上一时刻需要记忆的信息加上当前时刻需要记忆的信息。可以看到，单元状态 C_t 在单元的最上面一条线上流动，它代表了长期记忆，LSTM 正是通过这一点解决了长期记忆问题。



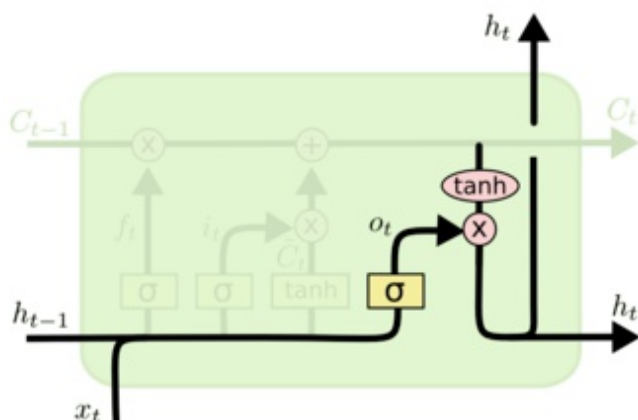
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

知乎 @随风

细胞状态更新

(3) 输出门

输出门用来计算哪些信息需要输出，输出为隐藏层状态 h_t 。从公式来看， h_t 表示对于当前时刻的状态信息 C_t （经过 tanh 变换），按照一定的比例输出。这里 h_t 与 RNN 中的 s_t 类似，代表短期记忆。



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

$$h_t = o_t * \tanh(C_t)$$

知乎 @随风

输出门 (output)

最终模型在 t 时刻的预测值 \tilde{y}_t 为: $\tilde{y}_t = \sigma(Wh_t + C)$, 其中 σ 为激活函数。LSTM 模型的参数较多, 包括 W_f 、 W_i 、 W_C 、 W_o 、 b_f 、 b_i 、 b_C 、 b_o , 类比于 RNN, 这 8 个参数是共享的。下面以 W_f 为例介绍 LSTM 的反向传播。由于序列在每个时刻都有损失函数, 因此最终的损失 L 为: $L = \sum_{t=1}^n L_t$, 以 $t = 2$ 时刻为例。

$$\begin{aligned} \frac{\partial L_2}{\partial W_f} = & \frac{\partial L_2}{\partial \tilde{y}_2} \frac{\partial \tilde{y}_2}{\partial h_2} \left(\frac{\partial h_2}{\partial o_2} \frac{\partial o_2}{\partial h_1} \frac{\partial h_1}{\partial \tanh(C_1)} \frac{\partial \tanh(C_1)}{\partial C_1} \frac{\partial C_1}{\partial f_1} \frac{\partial f_1}{\partial w_f} \right. \\ & + \frac{\partial h_2}{\partial \tanh(C_2)} \frac{\partial \tanh(C_2)}{\partial C_2} \left(\frac{\partial C_2}{\partial f_2} \left(\frac{\partial f_2}{\partial w_f} + \frac{\partial f_2}{\partial h_1} \frac{\partial h_1}{\partial \tanh(C_1)} \frac{\partial \tanh(C_1)}{\partial C_1} \frac{\partial C_1}{\partial f_1} \frac{\partial f_1}{\partial W_f} \right) \right. \\ & + \frac{\partial C_2}{\partial C_1} \frac{\partial C_1}{\partial f_1} \frac{\partial f_1}{\partial W_f} + \frac{\partial C_2}{\partial i_2} \frac{\partial i_2}{\partial h_1} \frac{\partial h_1}{\partial \tanh(C_1)} \frac{\partial \tanh(C_1)}{\partial C_1} \frac{\partial C_1}{\partial f_1} \frac{\partial f_1}{\partial W_f} \\ & \left. \left. + \frac{\partial C_2}{\partial \tilde{C}_2} \frac{\partial \tilde{C}_2}{\partial h_1} \frac{\partial h_1}{\partial \tanh(C_1)} \frac{\partial \tanh(C_1)}{\partial C_1} \frac{\partial C_1}{\partial f_1} \frac{\partial f_1}{\partial W_f} \right) \right) \end{aligned}$$

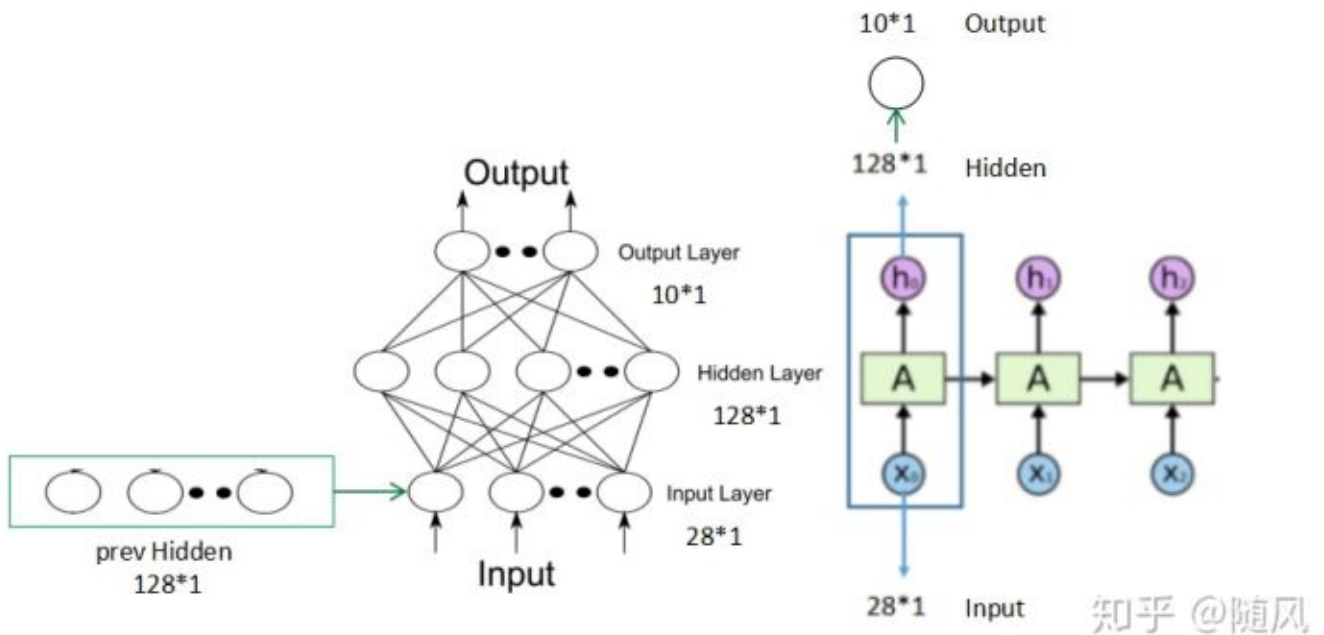
知乎 @随风

t=2 时刻链式求导

这里看到 LSTM 与 RNN 的区别了吗, 出现了很多累加的情况。如果自己推导一下, 会发现这里的链式求导法则“并不顺利”, 在 RNN 中, 基本上是一路连乘顺着写下去, 而在 LSTM 中, 出现了很多“分支”, 每连乘几步, 就出现累加的情况, 这里说一点 C_t 对于累加行为的贡献很大, 从公式看它的计算参数都直接或间接与 W_f 有关系。随着 t 的增大, 梯度公式越来越深, 而累加项也越来越多, 正是由于这样的特性, LSTM 解决了 RNN 中梯度消失的问题。

3.2 LSTM 信息流

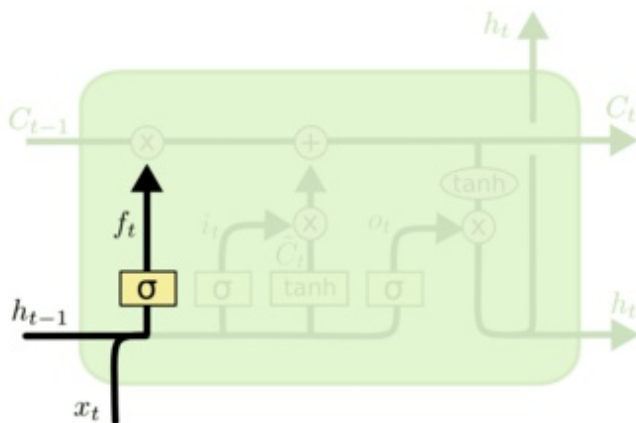
如下图所示, 假设在我们的训练数据中, 每个样本 x 是 3×28 维的矩阵, 那么将这个样本的每一行当成一个输入, 通过 3 个时间步骤展开 LSTM, 在每一个 LSTM 单元中, 我们的输入是 28×1 的向量, 假设我们要求隐藏层的输出是 128×1 的向量, 输出层的输出是 10×1 的向量。LSTM 单元把 28×1 维的向量映射为 128×1 维的向量, 再把 128×1 维的向量映射为 10×1 维的向量。下一个 LSTM 单元会接收上一个单元隐藏层传递的 128×1 维的向量, 结合新的 28×1 维的输入向量, 连接之后再映射成一个新的隐藏层的 128×1 维的向量, 再映射成一个新的输出层的 10×1 维的向量, 就这样一直处理下去, 直到网络的最后一个 LSTM 单元的隐藏层输出一个 128×1 维的向量, 输出层输出一个 10×1 维的向量。



LSTM输入输出信息流

(1) 遗忘门信息流

h_{t-1} 维度是 128×1 , x_t 维度是 28×1 , 将 h_{t-1} 、 x_t 连接起来构成维度是 156×1 , W_f 维度是 128×156 , b_f 维度是 128×1 , 所以 f_t 维度是 128×1 。在遗忘门中, 权重矩阵参数个数为 $W_f(128 * 156) + b_f(128 * 1) = 20096$



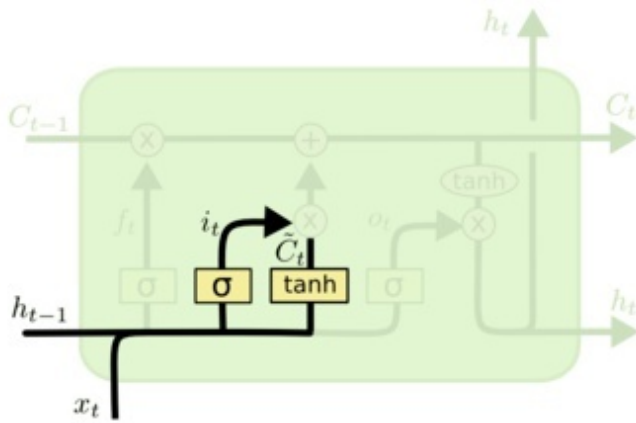
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

知乎 @随风

遗忘门 (forget)

(2) 输入门信息流

参照遗忘门的分析, W_i 维度是 128×156 , b_i 维度是 128×1 , 所以 i_t 维度是 128×1 。 W_C 维度是 128×156 , b_C 维度是 128×1 , 所以 \tilde{C}_t 维度是 128×1 。在输入门中, 权重矩阵参数个数为 $W_i(128 * 156) + b_i(128 * 1) + W_C(128 * 156) + b_C(128 * 1) = 40192$



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

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

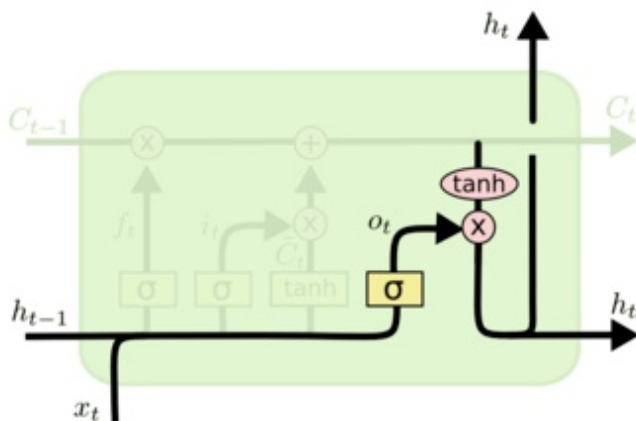
知乎 @随风

输入门 (input)

更新过程没有参数需要学习。

(3) 输出门信息流

参照上面的分析, W_o 维度是 128×156 , b_o 维度是 128×1 , 所以 o_t 维度是 128×1 。 C_t 的维度与 \tilde{C}_t 相同, 维度是 128×1 , 所以 h_t 维度是 128×1 。在输出门中, 权重矩阵参数个数为 $W_o(128 * 156) + b_o(128 * 1) = 20096$



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

$$h_t = o_t * \tanh(C_t)$$

知乎 @随风

输出门 (output)

输出门中的 h_t 是隐藏层的输出, 最后需要将 h_t 映射到输出层的输出 \tilde{y}_t :

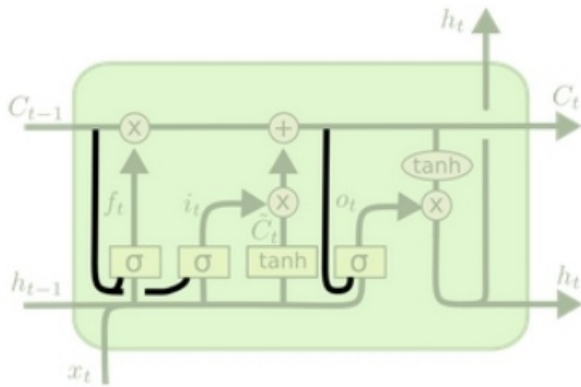
$\tilde{y}_t = \sigma(W h_t + C)$, W 的维度是 10×128 , C 的维度是 10×1 , 在输出层中, 权重矩阵参数个数为 $W(10 * 128) + C(10 * 1) = 1290$

现在我们可以计算 LSTM 的权重矩阵参数个数了, $20096 + 40192 + 20096 + 1290 = 81674$, 由于单元的参数共享, 所有的数据只会通过一个单元 (根据顺序流入一个单元的不同时刻), 然后不断更新它的权重。这里给出 LSTM 权重矩阵参数数量的计算公式:

$4((m + n)m + m) + m * k + k$, 其中 m 表示隐藏层神经元个数, n 表示输入神经元个数, k 表示输出神经元个数。

3.3 LSTM 的变形

(1) peephole (窥视孔) 连接：三个门不但依赖于输入 x_t 和上一时刻的隐状态 h_{t-1} ，也依赖于上一个时刻的内部记忆细胞状态 C_{t-1} 。



$$\begin{aligned} f_t &= \sigma(W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f) \\ i_t &= \sigma(W_i \cdot [C_{t-1}, h_{t-1}, x_t] + b_i) \\ o_t &= \sigma(W_o \cdot [C_t, h_{t-1}, x_t] + b_o) \end{aligned}$$

知乎 @随风

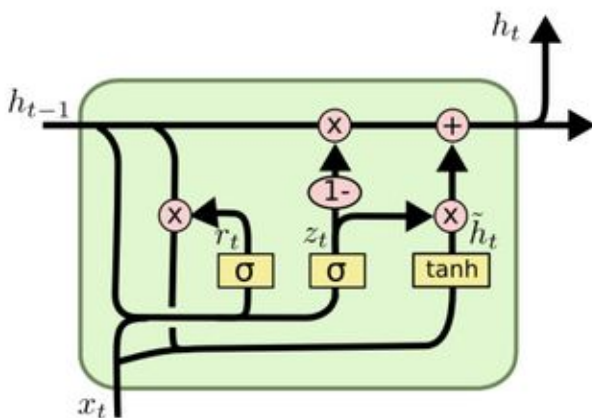
peephole 模型

(2) GRU (Gated Recurrent Unit)

输入门与遗忘门合并成一个门：更新门 [公式]

引入重置门 [公式]，用来控制输入候选状态 [公式] 的计算是否依赖上一时刻的状态 h_{t-1}

去除 LSTM 中的内部细胞记忆单元 C_t ，直接在当前状态 h_t 和历史状态 h_{t-1} 之间引入线性依赖关系



$$\begin{aligned} z_t &= \sigma(W_z \cdot [h_{t-1}, x_t]) \\ r_t &= \sigma(W_r \cdot [h_{t-1}, x_t]) \\ \tilde{h}_t &= \tanh(W \cdot [r_t * h_{t-1}, x_t]) \\ h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t \end{aligned}$$

知乎 @随风

GRU 模型

4 LSTM 实战

4.1 LSTM 参数调优

(1) 数据准备、预处理

神经网络具有很强的学习能力（也很容易过拟合），更适用于大规模数据集，因此需要准备大量、高质量并且带有干净标签的数据。从激活函数（sigmoid、tanh）可以看出，模型的输出绝对值一般在 0~1 之间，因此需要对数据进行归一化处理。常见的方法有：


1、Min-Max Normalization: [公式]

2、Average Normalization: [公式]

3、log function: [公式]

1、2 属于线性归一化，缺点是当有新数据加入时，可能导致 max 和 min 的变化，需要重新定义。3 属于非线性归一化，经常用在数据分化比较大的场景，有些数值很大，有些很小。

与归一化相近的概念是标准化：

Z-score规范化: [公式]

什么时候用归一化？什么时候用标准化？

- 1、如果对输出结果范围有要求，用归一化。
- 2、如果数据较为稳定，不存在极端的最大最小值，用归一化。
- 3、如果数据存在异常值和较多噪音，用标准化，可以间接通过中心化避免异常值和极端值的影响。

(2) 批处理

神经网络一般不会一个一个的训练样本，通常采用 minibatch 的方法一次训练一批数据，但不要使用过大的批处理，因为有可能导致低效和过拟合。

(3) 梯度归一化、梯度剪裁

因为采用了批处理，因此计算出来梯度之后，要除以 minibatch 的数量。如果训练 RNN 或者 LSTM，务必保证 gradient 的 norm 被约束在 5、10、15（前提还是要先归一化gradient），这一点在 RNN 和 LSTM 中很重要。在训练过程中，最好可以检查下梯度。

(4) 学习率

学习率是一个非常重要的参数，学习率太大将会导致训练过程非常不稳定甚至失败。太小将影响训练速度，通常设置为 0.1~0.001。

(5) 权值初始化

初始化参数对结果的影响至关重要，常见的初始化方法包括：

- 1、常量初始化：把权值或者偏置初始化为一个自定义的常数。
- 2、高斯分布初始化：需要给定高斯函数的均值与标准差。
- 3、xavier 初始化：对于均值为 0，方差为 $(1 / \text{输入的个数})$ 的均匀分布，如果我们更注重前向传播，可以选择 `fan_in`，即正向传播的输入个数；如果更注重反向传播，可以选择 `fan_out`，因为在反向传播的时候，`fan_out` 就是神经元的输入个数；如果两者都考虑，就选 $\text{average} = (\text{fan_in} + \text{fan_out}) / 2$ 。对于 Relu 激活函数，xavier 初始化很适合。

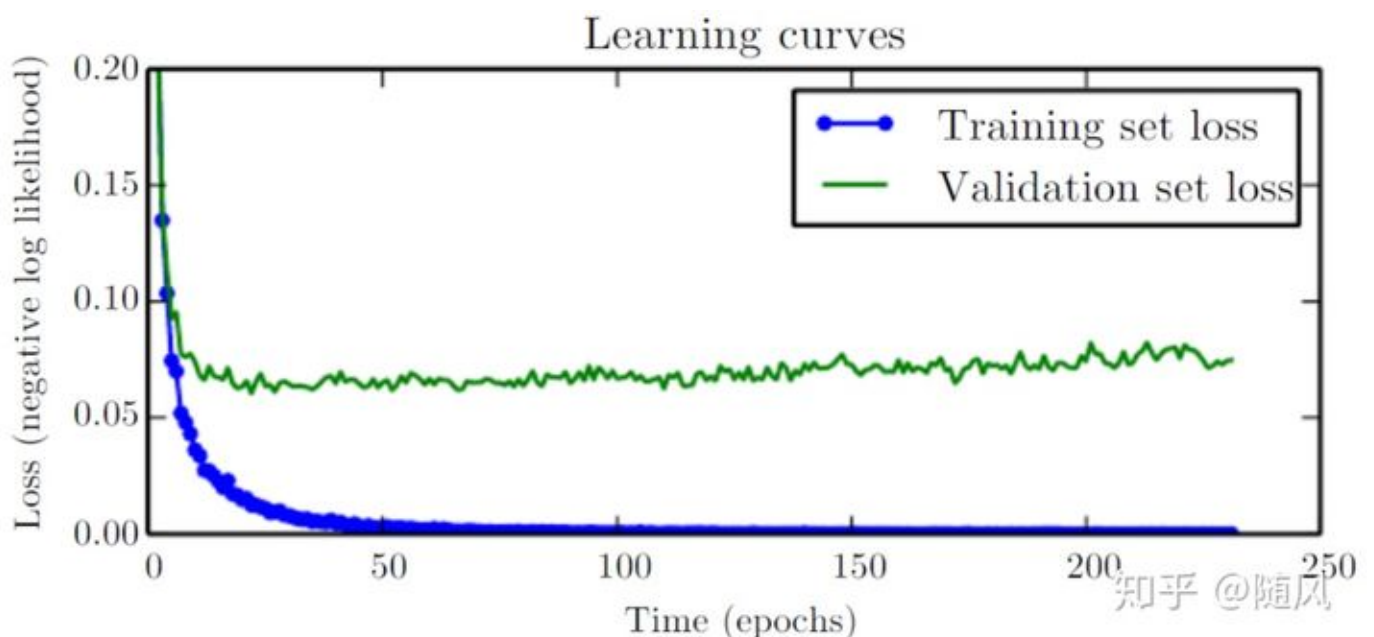
在权值初始化的时候，可以多尝试几种方法。此外，如果使用 LSTM 来解决长时依赖的问题，遗忘门初始化 bias 的时候要大一点（大于 1.0）。

(6) dropout

dropout 通过在训练的时候屏蔽部分神经元的方式，使网络最终具有较好的效果，相比于普通训练，需要花费更多的时间。记得在测试的时候关闭 dropout。**LSTM 的 dropout 只出现在同一时刻多层隐层之间，不会出现在不同时刻之间**（如果 dropout 跨越不同时刻，将导致随时间传递的状态信息丢失）。

(7) 提前终止

在训练的过程中，通常训练误差随着时间推移逐渐减小，而验证误差先减小后增大。期望的训练效果是：在训练集和验证集的效果都很好。训练集和验证集的 loss 都在下降，并且差不多在同个地方稳定下来。采用提前终止的方法，可以有效防止过拟合。



训练集、验证集误差

4.2 单时间序列 LSTM

类比于前面介绍的 ARIMA 模型，通过时间序列的先后关系进行预测。我们读取 user_balance_table.csv 文件，采用 Min-Max Normalization 方法对数据做归一化，将 2014-03-01~2014-07-31 的数据作为训练集，将 2014-08-01~2014-08-21 的数据作为验证集，将 2014-08-22~2014-08-31 的数据作为测试集。

```
import matplotlib.pyplot as plt
import pandas as pd
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler
import numpy as np

def generate_data():
    dateparse = lambda dates: pd.datetime.strptime(dates, '%Y%m%d')
    user_balance = pd.read_csv('./origin_data/user_balance_table.csv', parse_dates=['report_date'])
    user_balance.index = user_balance['report_date']

    user_balance = user_balance.groupby(['report_date'])['share_amt', 'total_purchase_amt'].reset_index(inplace=True)
    user_balance.index = user_balance['report_date']

    user_balance = user_balance['2014-03-01':'2014-08-31']

    data = {'total_purchase_amt': user_balance['total_purchase_amt']}

    df = pd.DataFrame(data=data, index=user_balance.index)
    df.to_csv(path_or_buf='./mid_data/single_purchase_seq.csv')

# 数据集归一化
def get_normal_data(purchase_seq):
    scaler = MinMaxScaler(feature_range=(0, 1))
    scaled_data = scaler.fit_transform(purchase_seq[['total_purchase_amt']])
    scaled_x_data = scaled_data[0: -1]
    scaled_y_data = scaled_data[1:]
    return scaled_x_data, scaled_y_data, scaler

# 构造训练集
def get_train_data(scaled_x_data, scaled_y_data, divide_train_valid_index, time_step):
    train_x, train_y = [], []
    normalized_train_feature = scaled_x_data[0: -divide_train_valid_index]
```

```

normalized_train_label = scaled_y_data[0: -divide_train_valid_index]
for i in range(len(normalized_train_feature) - time_step + 1):
    train_x.append(normalized_train_feature[i:i + time_step].tolist())
    train_y.append(normalized_train_label[i:i + time_step].tolist())
return train_x, train_y

```

构造拟合训练集

```

def get_train_fit_data(scaled_x_data, scaled_y_data, divide_train_valid_index, time_step):
    train_fit_x, train_fit_y = [], []
    normalized_train_feature = scaled_x_data[0: -divide_train_valid_index]
    normalized_train_label = scaled_y_data[0: -divide_train_valid_index]
    train_fit_remain = len(normalized_train_label) % time_step
    train_fit_num = int((len(normalized_train_label) - train_fit_remain) / time_step)
    temp = []
    for i in range(train_fit_num):
        train_fit_x.append(normalized_train_feature[i * time_step:(i + 1) * time_step].tolist())
        temp.extend(normalized_train_label[i * time_step:(i + 1) * time_step].tolist())
    if train_fit_remain > 0:
        train_fit_x.append(normalized_train_feature[-time_step:].tolist())
        temp.extend(normalized_train_label[-train_fit_remain:].tolist())
    for i in temp:
        train_fit_y.append(i[0])
    return train_fit_x, train_fit_y, train_fit_remain

```

构造验证集

```

def get_valid_data(scaled_x_data, scaled_y_data, divide_train_valid_index, divide_valid_index):
    valid_x, valid_y = [], []
    normalized_valid_feature = scaled_x_data[-divide_train_valid_index: -divide_valid_index]
    normalized_valid_label = scaled_y_data[-divide_train_valid_index: -divide_valid_index]
    valid_remain = len(normalized_valid_label) % time_step
    valid_num = int((len(normalized_valid_label) - valid_remain) / time_step)
    temp = []
    for i in range(valid_num):
        valid_x.append(normalized_valid_feature[i * time_step:(i + 1) * time_step].tolist())
        temp.extend(normalized_valid_label[i * time_step:(i + 1) * time_step].tolist())
    if valid_remain > 0:
        valid_x.append(normalized_valid_feature[-time_step:].tolist())
        temp.extend(normalized_valid_label[-valid_remain:].tolist())
    for i in temp:
        valid_y.append(i[0])
    return valid_x, valid_y, valid_remain

```

构造测试集

```
def get_test_data(scaled_x_data, scaled_y_data, divide_valid_test_index, time_step):
    test_x, test_y = [], []
    normalized_test_feature = scaled_x_data[-divide_valid_test_index:]
    normalized_test_label = scaled_y_data[-divide_valid_test_index:]
    test_remain = len(normalized_test_label) % time_step
    test_num = int((len(normalized_test_label) - test_remain) / time_step)
    temp = []
    for i in range(test_num):
        test_x.append(normalized_test_feature[i * time_step:(i + 1) * time_step].tolist())
        temp.extend(normalized_test_label[i * time_step:(i + 1) * time_step].tolist())
    if test_remain > 0:
        test_x.append(scaled_x_data[-time_step:].tolist())
        temp.extend(normalized_test_label[-test_remain:].tolist())
    for i in temp:
        test_y.append(i[0])
    return test_x, test_y, test_remain
```

generate_data()



训练 LSTM 模型。前面几章介绍的模型，只要参数确定，无论执行多少次结果都一样。不同于前面几章的模型，这里的权值矩阵采用截断高斯初始化，因此每次训练模型产生的结果都不太一样，最后的结果图是经过多次训练，选择了一个比较好的模型。

```
import matplotlib.pyplot as plt
import pandas as pd
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler
import numpy as np
```

模型参数

```
lr = 1e-3 # 学习率
batch_size = 10 # minibatch 大小
rnn_unit = 30 # LSTM 隐藏层神经元数量
input_size = 1 # 单元的输入数量
output_size = 1 # 单元的输出数量
time_step = 15 # 时间长度
epochs = 1000 # 训练次数
gradient_threshold = 15 # 梯度裁剪阈值
stop_loss = np.float32(0.045) # 训练停止条件。当训练误差 + 验证误差小于阈值时，停止训练
train_keep_prob = [1.0, 0.5, 1.0] # 训练时 dropout 神经元保留比率
```

数据切分参数

divide_train_valid_index = 30

divide_valid_test_index = 10

数据准备

dateparse = lambda dates: pd.datetime.strptime(dates, '%Y-%m-%d')

single_purchase_seq = pd.read_csv('../mid_data/single_purchase_seq.csv', parse_dates=[

scaled_x_data, scaled_y_data, scaler = get_normal_data(single_purchase_seq)

train_x, train_y = get_train_data(scaled_x_data, scaled_y_data, divide_train_valid_ind

train_fit_x, train_fit_y, train_fit_remain = get_train_fit_data(scaled_x_data, scaled_

valid_x, valid_y, valid_remain = get_valid_data(scaled_x_data, scaled_y_data, divide_t

test_x, test_y, test_remain = get_test_data(scaled_x_data, scaled_y_data, divide_valid

def lstm(X, keep_prob):

batch_size = tf.shape(X)[0] # minibatch 大小

输入到 LSTM 输入的转换, 一层全连接的网络, 其中权重初始化采用截断的高斯分布, 激活函数采用t

weights = tf.Variable(tf.truncated_normal(shape=[input_size, rnn_unit]))

biases = tf.Variable(tf.constant(0.1, shape=[rnn_unit,]))

input = tf.reshape(X, [-1, input_size])

tanh_layer = tf.nn.tanh(tf.matmul(input, weights) + biases)

input_rnn = tf.nn.dropout(tanh_layer, keep_prob[0])

input_rnn = tf.reshape(input_rnn, [-1, time_step, rnn_unit])

两层 LSTM 网络, 激活函数默认采用 tanh, 当网络层数较深时, 建议使用 relu

initializer = tf.truncated_normal_initializer()

cell_1 = tf.nn.rnn_cell.LSTMCell(forget_bias=1.0, num_units=rnn_unit, use_peephole

cell_1_drop = tf.nn.rnn_cell.DropoutWrapper(cell=cell_1, output_keep_prob=keep_pro

cell_2 = tf.nn.rnn_cell.LSTMCell(forget_bias=1.0, num_units=rnn_unit, use_peephole

cell_2_drop = tf.nn.rnn_cell.DropoutWrapper(cell=cell_2, output_keep_prob=keep_pro

mutilstm_cell = tf.nn.rnn_cell.MultiRNNCell(cells=[cell_1_drop, cell_2_drop], stat

init_state = mutilstm_cell.zero_state(batch_size, dtype=tf.float32)

with tf.variable_scope('lstm', reuse=tf.AUTO_REUSE):

output, state = tf.nn.dynamic_rnn(cell=mutilstm_cell, inputs=input_rnn, initia

return output, state

获取拟合数据, 这里用于拟合, 关闭 dropout

def get_fit_seq(x, remain, sess, output, X, keep_prob, scaler, inverse):


```

fit_seq = []
if inverse:
    # 前面对数据进行了归一化, 这里反归一化还原数据
    temp = []
    for i in range(len(x)):
        next_seq = sess.run(output, feed_dict={X: [x[i]], keep_prob: [1.0, 1.0, 1.0]})
        if i == len(x) - 1:
            temp.extend(scaler.inverse_transform(next_seq[0].reshape(-1, 1))[-remain:])
        else:
            temp.extend(scaler.inverse_transform(next_seq[0].reshape(-1, 1)))
    for i in temp:
        fit_seq.append(i[0])
else:
    for i in range(len(x)):
        next_seq = sess.run(output,
                              feed_dict={X: [x[i]], keep_prob: [1.0, 1.0, 1.0]})
        if i == len(x) - 1:
            fit_seq.extend(next_seq[0].reshape(1, -1).tolist()[0][-remain:])
        else:
            fit_seq.extend(next_seq[0].reshape(1, -1).tolist()[0])

return fit_seq

```

```

def train_lstm():
    X = tf.placeholder(tf.float32, [None, time_step, input_size])
    Y = tf.placeholder(tf.float32, [None, time_step, output_size])

    keep_prob = tf.placeholder(tf.float32, [None])
    output, state = lstm(X, keep_prob)
    loss = tf.losses.mean_squared_error(tf.reshape(output, [-1]), tf.reshape(Y, [-1]))

    # 梯度优化与裁剪
    optimizer = tf.train.AdamOptimizer(learning_rate=lr)
    grads, variables = zip(*optimizer.compute_gradients(loss))
    grads, global_norm = tf.clip_by_global_norm(grads, gradient_threshold)
    train_op = optimizer.apply_gradients(zip(grads, variables))

    X_train_fit = tf.placeholder(tf.float32, [None])
    Y_train_fit = tf.placeholder(tf.float32, [None])
    train_fit_loss = tf.losses.mean_squared_error(tf.reshape(X_train_fit, [-1]), tf.reshape(Y_train_fit, [-1]))

    X_valid = tf.placeholder(tf.float32, [None])
    Y_valid = tf.placeholder(tf.float32, [None])
    valid_fit_loss = tf.losses.mean_squared_error(tf.reshape(X_valid, [-1]), tf.reshape(Y_valid, [-1]))

```

```

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    fit_loss_seq = []
    valid_loss_seq = []

    for epoch in range(epochs):
        for index in range(len(train_x) - batch_size + 1):
            sess.run(train_op, feed_dict={X: train_x[index: index + batch_size], Y:

# 拟合训练集和验证集
train_fit_seq = get_fit_seq(train_fit_x, train_fit_remain, sess, output, X
train_loss = sess.run(train_fit_loss, {X_train_fit: train_fit_seq, Y_train
fit_loss_seq.append(train_loss)

valid_seq = get_fit_seq(valid_x, valid_remain, sess, output, X, keep_prob,
valid_loss = sess.run(valid_fit_loss, {X_valid: valid_seq, Y_valid: valid_
valid_loss_seq.append(valid_loss)

print('epoch:', epoch + 1, 'fit loss:', train_loss, 'valid loss:', valid_l

# 提前终止条件。
# 常见的方法是验证集达到最小值，再往后训练 n 步，loss 不再减小，实际测试这里使用的
# 这里选择 stop_loss 是经过多次尝试得到的阈值。
if train_loss + valid_loss <= stop_loss:
    train_fit_seq = get_fit_seq(train_fit_x, train_fit_remain, sess, output
    valid_fit_seq = get_fit_seq(valid_x, valid_remain, sess, output, X, ke
    test_fit_seq = get_fit_seq(test_x, test_remain, sess, output, X, keep_
    print('best epoch: ', epoch + 1)
    break

return fit_loss_seq, valid_loss_seq, train_fit_seq, valid_fit_seq, test_fit_seq

fit_loss_seq, valid_loss_seq, train_fit_seq, valid_fit_seq, test_fit_seq = train_lstm(

# 切分训练集、测试集
purchase_seq_train = single_purchase_seq[1:-divide_train_valid_index]
purchase_seq_valid = single_purchase_seq[-divide_train_valid_index:-divide_valid_test_
purchase_seq_test = single_purchase_seq[-divide_valid_test_index:]

plt.figure(figsize=(18, 12))

plt.subplot(221)

```

```

plt.title('loss')
plt.plot(fit_loss_seq, label='fit_loss', color='blue')
plt.plot(valid_loss_seq, label='valid_loss', color='red')
plt.legend(loc='best')

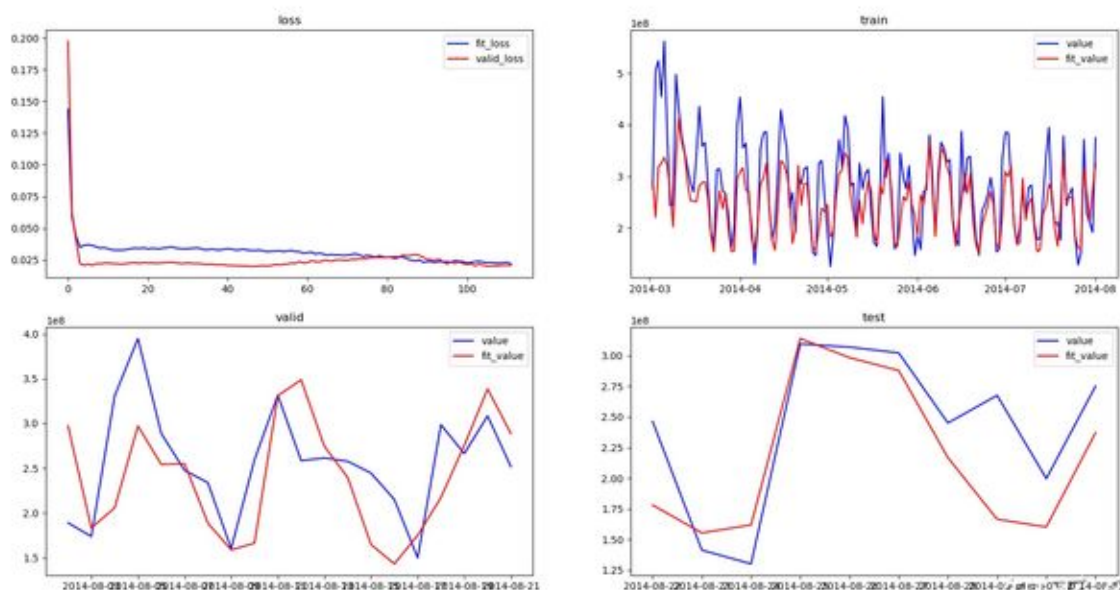
plt.subplot(222)
plt.title('train')
seq_train_fit = pd.DataFrame(columns=['total_purchase_amt'], data=train_fit_seq, index=
plt.plot(purchase_seq_train['total_purchase_amt'], label='value', color='blue')
plt.plot(seq_train_fit['total_purchase_amt'], label='fit_value', color='red')
plt.legend(loc='best')

plt.subplot(223)
plt.title('valid')
seq_valid_fit = pd.DataFrame(columns=['total_purchase_amt'], data=valid_fit_seq, index=
plt.plot(purchase_seq_valid['total_purchase_amt'], label='value', color='blue')
plt.plot(seq_valid_fit['total_purchase_amt'], label='fit_value', color='red')
plt.legend(loc='best')

plt.subplot(224)
plt.title('test')
seq_test_fit = pd.DataFrame(columns=['total_purchase_amt'], data=test_fit_seq, index=p
plt.plot(purchase_seq_test['total_purchase_amt'], label='value', color='blue')
plt.plot(seq_test_fit['total_purchase_amt'], label='fit_value', color='red')
plt.legend(loc='best')

plt.show()

```



知乎@随风

4.3 多时间序列 LSTM

类比于前面介绍的多元回归模型，通过不同时间序列之间的关系进行预测。选取的特征序列：share_amt（余额宝今日收益）、mfd_daily_yield（余额宝万份收益）、mfd_7daily_yield（余额宝七日年化收益率）、Interest_O_N（银行隔夜利率）、Interest_1_W（银行1周利率）、Interest_1_M（银行1个月利率）。这些特征列存在于 user_balance_table.csv、mfd_day_share_interest.csv、mfd_bank_shibor 这三个文件中。

我们读取这三个文件，采用 Min-Max Normalization 方法对数据做归一化，将2014-03-01~2014-07-31 的数据作为训练集，将 2014-08-01~2014-08-21 的数据作为验证集，将 2014-08-22~2014-08-31 的数据作为测试集。

```
import matplotlib.pyplot as plt
import pandas as pd
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler
import numpy as np

def generate_data():
    dateparse = lambda dates: pd.datetime.strptime(dates, '%Y%m%d')
    user_balance = pd.read_csv('../origin_data/user_balance_table.csv', parse_dates=[
        'report_date'], date_parser=dateparse)
    user_balance.index = user_balance['report_date']

    mfd_day_share_interest = pd.read_csv('../origin_data/mfd_day_share_interest.csv',
        date_parser=dateparse)
    mfd_day_share_interest.index = mfd_day_share_interest['mfd_date']

    mfd_bank_shibor = pd.read_csv('../origin_data/mfd_bank_shibor.csv', parse_dates=[
        'mfd_date'], date_parser=dateparse)
    mfd_bank_shibor.index = mfd_bank_shibor['mfd_date']

    user_balance = user_balance.groupby(['report_date'])['share_amt', 'total_purchase_amt'].reset_index(inplace=True)
    user_balance.index = user_balance['report_date']

    user_balance = user_balance['2014-03-01':'2014-08-31']
    mfd_day_share_interest = mfd_day_share_interest['2014-03-01':'2014-08-31']
    mfd_bank_shibor = mfd_bank_shibor['2014-03-01':'2014-08-31']

    data = {'share_amt': user_balance['share_amt'],
            'total_purchase_amt': user_balance['total_purchase_amt'],
```

```

'mfd_daily_yield': mfd_day_share_interest['mfd_daily_yield'],
'mfd_7daily_yield': mfd_day_share_interest['mfd_7daily_yield'],
'Interest_0_N': mfd_bank_shibor['Interest_0_N'],
'Interest_1_W': mfd_bank_shibor['Interest_1_W'],
'Interest_1_M': mfd_bank_shibor['Interest_1_M']}

```

```
df = pd.DataFrame(data=data, index=user_balance.index)
```

```

df['Interest_0_N'].fillna(df['Interest_0_N'].median(), inplace=True)
df['Interest_1_W'].fillna(df['Interest_1_W'].median(), inplace=True)
df['Interest_1_M'].fillna(df['Interest_1_M'].median(), inplace=True)

```

```
df.to_csv(path_or_buf='../mid_data/multi_purchase_seq.csv')
```

数据集归一化

```

def get_normal_data(purchase_seq):
    scaler_x = MinMaxScaler(feature_range=(0, 1))
    scaler_y = MinMaxScaler(feature_range=(0, 1))
    scaled_x_data = scaler_x.fit_transform(purchase_seq[['Interest_1_M', 'Interest_1_
scaled_y_data = scaler_y.fit_transform(purchase_seq[['total_purchase_amt']])
    return scaled_x_data, scaled_y_data, scaler_y

```

构造训练集

```

def get_train_data(scaled_x_data, scaled_y_data, divide_train_valid_index, time_step):
    train_x, train_y = [], []
    normalized_train_feature = scaled_x_data[0: -divide_train_valid_index]
    normalized_train_label = scaled_y_data[0: -divide_train_valid_index]
    for i in range(len(normalized_train_feature) - time_step + 1):
        train_x.append(normalized_train_feature[i:i + time_step].tolist())
        train_y.append(normalized_train_label[i:i + time_step].tolist())
    return train_x, train_y

```

构造拟合训练集

```

def get_train_fit_data(scaled_x_data, scaled_y_data, divide_train_valid_index, time_st
    train_fit_x, train_fit_y = [], []
    normalized_train_feature = scaled_x_data[0: -divide_train_valid_index]
    normalized_train_label = scaled_y_data[0: -divide_train_valid_index]
    train_fit_remain = len(normalized_train_label) % time_step
    train_fit_num = int((len(normalized_train_label) - train_fit_remain) / time_step)
    temp = []
    for i in range(train_fit_num):
        train_fit_x.append(normalized_train_feature[i * time_step:(i + 1) * time_step]

```

```

    temp.extend(normalized_train_label[i * time_step:(i + 1) * time_step].tolist())
if train_fit_remain > 0:
    train_fit_x.append(normalized_train_feature[-time_step:].tolist())
    temp.extend(normalized_train_label[-train_fit_remain:].tolist())
for i in temp:
    train_fit_y.append(i[0])
return train_fit_x, train_fit_y, train_fit_remain

```

构造验证集

```

def get_valid_data(scaled_x_data, scaled_y_data, divide_train_valid_index, divide_vali
    valid_x, valid_y = [], []
    normalized_valid_feature = scaled_x_data[-divide_train_valid_index: -divide_valid_
    normalized_valid_label = scaled_y_data[-divide_train_valid_index: -divide_valid_te
    valid_remain = len(normalized_valid_label) % time_step
    valid_num = int((len(normalized_valid_label) - valid_remain) / time_step)
    temp = []
    for i in range(valid_num):
        valid_x.append(normalized_valid_feature[i * time_step:(i + 1) * time_step].tol
        temp.extend(normalized_valid_label[i * time_step:(i + 1) * time_step].tolist())
    if valid_remain > 0:
        valid_x.append(normalized_valid_feature[-time_step:].tolist())
        temp.extend(normalized_valid_label[-valid_remain:].tolist())
    for i in temp:
        valid_y.append(i[0])
    return valid_x, valid_y, valid_remain

```

构造测试集

```

def get_test_data(scaled_x_data, scaled_y_data, divide_valid_test_index, time_step):
    test_x, test_y = [], []
    normalized_test_feature = scaled_x_data[-divide_valid_test_index:]
    normalized_test_label = scaled_y_data[-divide_valid_test_index:]
    test_remain = len(normalized_test_label) % time_step
    test_num = int((len(normalized_test_label) - test_remain) / time_step)
    temp = []
    for i in range(test_num):
        test_x.append(normalized_test_feature[i * time_step:(i + 1) * time_step].tolis
        temp.extend(normalized_test_label[i * time_step:(i + 1) * time_step].tolist())
    if test_remain > 0:
        test_x.append(scaled_x_data[-time_step:].tolist())
        temp.extend(normalized_test_label[-test_remain:].tolist())
    for i in temp:
        test_y.append(i[0])
    return test_x, test_y, test_remain

```



```
generate_data()
```

训练 LSTM 模型。大体流程与单时间序列 LSTM 一致，只是这里的输入序列变成了 6 个，因为输入的变化，相应模型参数的取值也发生了变化。

```
import matplotlib.pyplot as plt
import pandas as pd
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler
import numpy as np

# 模型参数
lr = 1e-3 # 学习率
batch_size = 10 # minibatch大小
rnn_unit = 40 # LSTM 隐藏层神经元数量
input_size = 6 # 单元的输入数量
output_size = 1 # 单元的输出数量
time_step = 15 # 时间长度
epochs = 1000 # 训练次数
gradient_threshold = 5 # 梯度裁剪阈值
stop_loss = np.float32(0.025) # 训练停止条件。当训练误差 + 验证误差小于阈值时，停止训练
train_keep_prob = [1.0, 0.5, 1.0] # 训练时 dropout 神经元保留比率

# 数据切分参数
divide_train_valid_index = 31
divide_valid_test_index = 10

# 数据准备
dateparse = lambda dates: pd.datetime.strptime(dates, '%Y-%m-%d')
purchase_seq = pd.read_csv('../mid_data/multi_purchase_seq.csv', parse_dates=['report_

scaled_x_data, scaled_y_data, scaler_y = get_normal_data(purchase_seq)
train_x, train_y = get_train_data(scaled_x_data, scaled_y_data, divide_train_valid_ind
train_fit_x, train_fit_y, train_fit_remain = get_train_fit_data(scaled_x_data, scaled_
valid_x, valid_y, valid_remain = get_valid_data(scaled_x_data, scaled_y_data, divide_t
test_x, test_y, test_remain = get_test_data(scaled_x_data, scaled_y_data, divide_valid

def lstm(X, keep_prob):
    batch_size = tf.shape(X)[0] # minibatch 大小
```

```

# 输入到 LSTM 输入层的转换, 一层全连接的网络, 其中权重初始化采用截断的高斯分布, 激活函数采用tanh
weights = tf.Variable(tf.truncated_normal(shape=[input_size, rnn_unit]))
biases = tf.Variable(tf.constant(0.1, shape=[rnn_unit, ]))
input = tf.reshape(X, [-1, input_size])

input_layer = tf.nn.tanh(tf.matmul(input, weights) + biases)
input_rnn = tf.nn.dropout(input_layer, keep_prob[0])
input_rnn = tf.reshape(input_rnn, [-1, time_step, rnn_unit])

# 两层 LSTM 网络, 激活函数默认采用 tanh, 当网络层数较深时, 建议使用 relu
initializer = tf.truncated_normal_initializer()
cell_1 = tf.nn.rnn_cell.LSTMCell(forget_bias=1.0, num_units=rnn_unit, use_peephole=False)
cell_1_drop = tf.nn.rnn_cell.DropoutWrapper(cell=cell_1, output_keep_prob=keep_prob[0])

cell_2 = tf.nn.rnn_cell.LSTMCell(forget_bias=1.0, num_units=rnn_unit, use_peephole=False)
cell_2_drop = tf.nn.rnn_cell.DropoutWrapper(cell=cell_2, output_keep_prob=keep_prob[0])

mutilstm_cell = tf.nn.rnn_cell.MultiRNNCell(cells=[cell_1_drop, cell_2_drop], state_initializer=initializer)
init_state = mutilstm_cell.zero_state(batch_size, dtype=tf.float32)

with tf.variable_scope('lstm', reuse=tf.AUTO_REUSE):
    output, state = tf.nn.dynamic_rnn(cell=mutilstm_cell, inputs=input_rnn, initial_state=init_state)

return output, state

# 获取拟合数据, 这里用于拟合, 关闭 dropout
def get_fit_seq(x, remain, sess, output, X, keep_prob, scaler, inverse):
    fit_seq = []
    if inverse:
        # 前面对数据进行了归一化, 这里反归一化还原数据
        temp = []
        for i in range(len(x)):
            next_seq = sess.run(output, feed_dict={X: [x[i]], keep_prob: [1.0, 1.0, 1.0]})
            if i == len(x) - 1:
                temp.extend(scaler.inverse_transform(next_seq[0].reshape(-1, 1))[-remain:])
            else:
                temp.extend(scaler.inverse_transform(next_seq[0].reshape(-1, 1)))
        for i in temp:
            fit_seq.append(i[0])
    else:
        for i in range(len(x)):
            next_seq = sess.run(output, feed_dict={X: [x[i]], keep_prob: [1.0, 1.0, 1.0]})
            if i == len(x) - 1:
                fit_seq.extend(next_seq[0].reshape(1, -1).tolist()[0][-remain:])

```

```

    else:
        fit_seq.extend(next_seq[0].reshape(1, -1).tolist()[0])

    return fit_seq

def train_lstm():
    X = tf.placeholder(tf.float32, [None, time_step, input_size])
    Y = tf.placeholder(tf.float32, [None, time_step, output_size])

    keep_prob = tf.placeholder(tf.float32, [None])
    output, state = lstm(X, keep_prob)
    loss = tf.losses.mean_squared_error(tf.reshape(output, [-1]), tf.reshape(Y, [-1]))

    # 梯度优化与裁剪
    optimizer = tf.train.AdamOptimizer(learning_rate=lr)
    grads, variables = zip(*optimizer.compute_gradients(loss))
    grads, global_norm = tf.clip_by_global_norm(grads, gradient_threshold)
    train_op = optimizer.apply_gradients(zip(grads, variables))

    X_train_fit = tf.placeholder(tf.float32, [None])
    Y_train_fit = tf.placeholder(tf.float32, [None])
    train_fit_loss = tf.losses.mean_squared_error(tf.reshape(X_train_fit, [-1]), tf.re

    X_valid = tf.placeholder(tf.float32, [None])
    Y_valid = tf.placeholder(tf.float32, [None])
    valid_fit_loss = tf.losses.mean_squared_error(tf.reshape(X_valid, [-1]), tf.reshap

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        fit_loss_seq = []
        valid_loss_seq = []

        for epoch in range(epochs):
            for index in range(len(train_x) - batch_size + 1):
                sess.run(train_op, feed_dict={X: train_x[index: index + batch_size], Y:

            # 拟合训练集和验证集
            train_fit_seq = get_fit_seq(train_fit_x, train_fit_remain, sess, output, X
            train_loss = sess.run(train_fit_loss, {X_train_fit: train_fit_seq, Y_train
            fit_loss_seq.append(train_loss)

            valid_seq = get_fit_seq(valid_x, valid_remain, sess, output, X, keep_prob,
            valid_loss = sess.run(valid_fit_loss, {X_valid: valid_seq, Y_valid: valid
            valid_loss_seq.append(valid_loss)

```

```

print('epoch:', epoch + 1, 'fit loss:', train_loss, 'valid loss:', valid_l

# 提前终止条件。
# 常见的方法是验证集达到最小值，再往后训练 n 步，loss 不再减小，实际测试这里使用的
# 这里选择 stop_loss 是经过多次尝试得到的阈值。
if train_loss + valid_loss <= stop_loss:
    train_fit_seq = get_fit_seq(train_fit_x, train_fit_remain, sess, output
    valid_fit_seq = get_fit_seq(valid_x, valid_remain, sess, output, X, ke
    test_fit_seq = get_fit_seq(test_x, test_remain, sess, output, X, keep_
    print('best epoch: ', epoch + 1)
    break

return fit_loss_seq, valid_loss_seq, train_fit_seq, valid_fit_seq, test_fit_seq

fit_loss_seq, valid_loss_seq, train_fit_seq, valid_fit_seq, test_fit_seq = train_lstm(

# 切分训练集、验证集、测试集
purchase_seq_train = purchase_seq[0:-divide_train_valid_index]
purchase_seq_valid = purchase_seq[-divide_train_valid_index:-divide_valid_test_index]
purchase_seq_test = purchase_seq[-divide_valid_test_index:]

plt.figure(figsize=(18, 12))

plt.subplot(221)
plt.title('loss')
plt.plot(fit_loss_seq, label='fit_loss', color='blue')
plt.plot(valid_loss_seq, label='valid_loss', color='red')
plt.legend(loc='best')

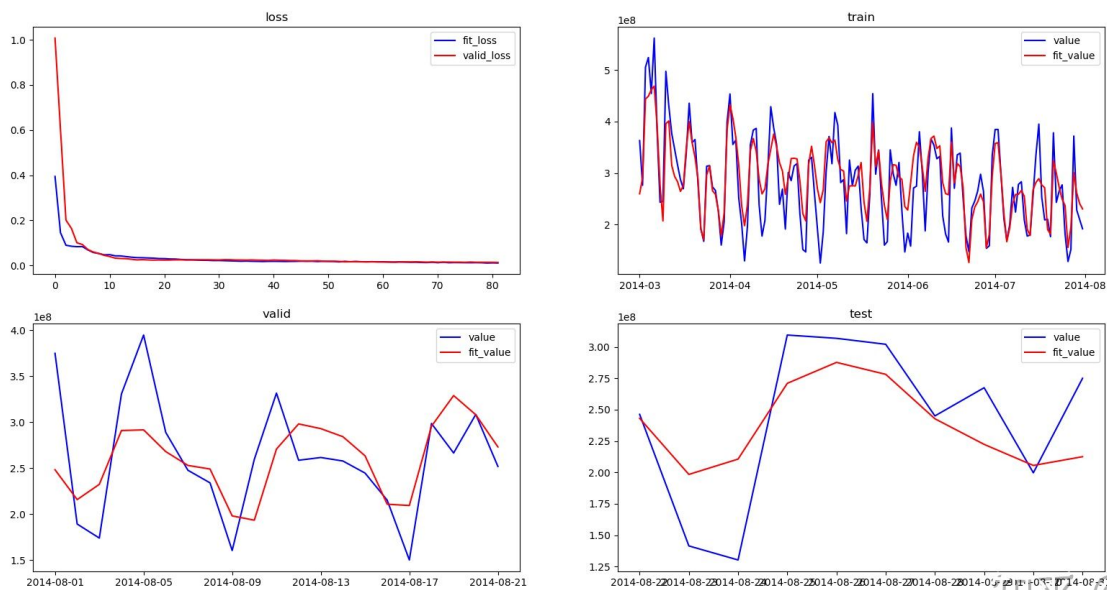
plt.subplot(222)
plt.title('train')
seq_train_fit = pd.DataFrame(columns=['total_purchase_amt'], data=train_fit_seq, index
plt.plot(purchase_seq_train['total_purchase_amt'], label='value', color='blue')
plt.plot(seq_train_fit['total_purchase_amt'], label='fit_value', color='red')
plt.legend(loc='best')

plt.subplot(223)
plt.title('valid')
seq_valid_fit = pd.DataFrame(columns=['total_purchase_amt'], data=valid_fit_seq, index
plt.plot(purchase_seq_valid['total_purchase_amt'], label='value', color='blue')
plt.plot(seq_valid_fit['total_purchase_amt'], label='fit_value', color='red')
plt.legend(loc='best')

```

```
plt.subplot(224)
plt.title('test')
seq_test_fit = pd.DataFrame(columns=['total_purchase_amt'], data=test_fit_seq, index=p
plt.plot(purchase_seq_test['total_purchase_amt'], label='value', color='blue')
plt.plot(seq_test_fit['total_purchase_amt'], label='fit_value', color='red')
plt.legend(loc='best')

plt.show()
```







多时间序列 LSTM 效果

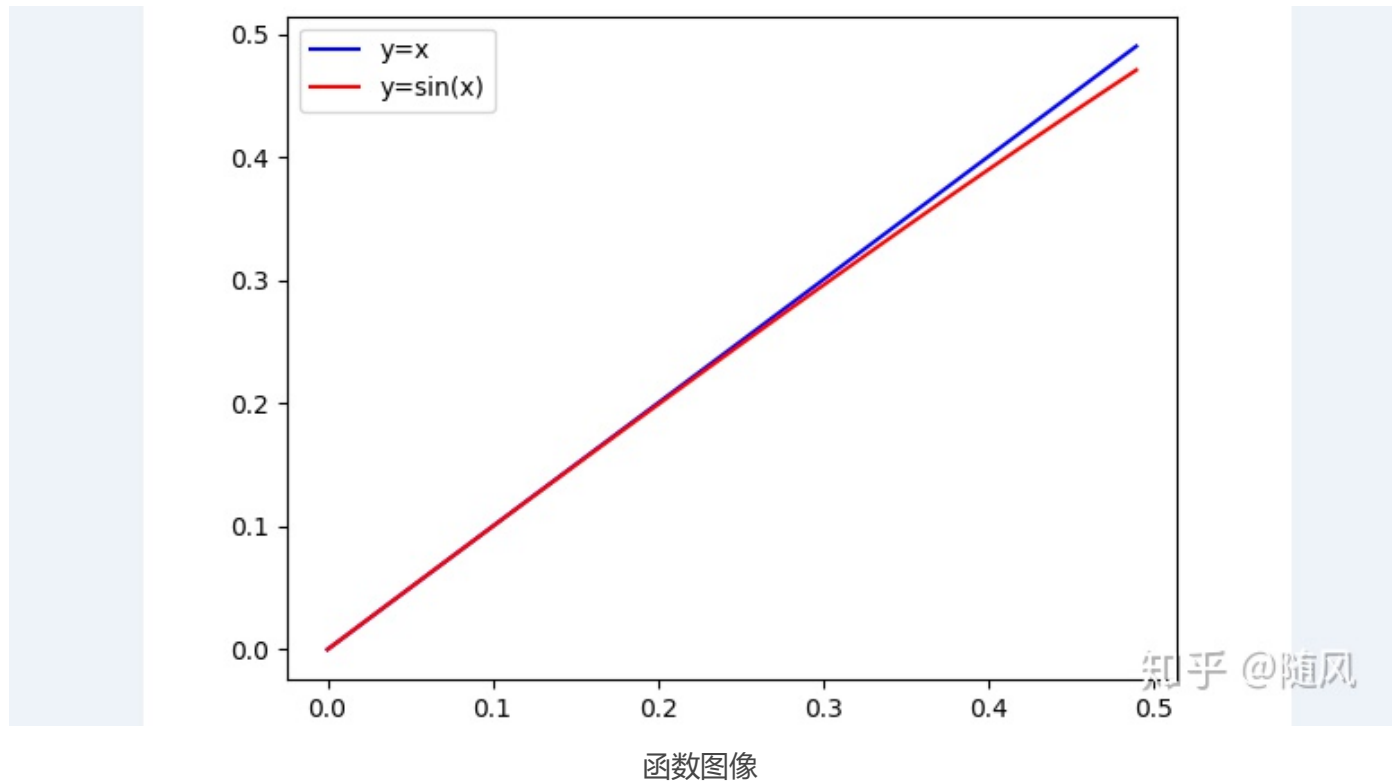
由于数据集的限制，这里并没有选择与目标序列相关性最强的特征序列（例如股票序列）。显然，更好的特征序列对模型提升影响极大。另一点需要指出，假如今天是 8 月 21 号，你需要对 8 月 22 号到 31 号的数据做预测，你也并不知道特征序列的取值，也就无法预测。

5 总结

5.1 线性与非线性模型应用于同一时间序列

首先来比较一下 ARIMA、单时间序列 LSTM，**ARIMA 从线性自相关的角度进行建模，单时间序列 LSTM 从非线性自相关的角度进行建模**。再来比较一下多元 LR、多时间序列 LSTM，**多元 LR 从线性互相关的角度进行建模，多时间序列 LSTM 从非线性互相关的角度进行建模**。这里有个疑问，为什么我们既可以通过线性相关建模，又可以通过非线性相关建模？

先要明确模型的意义，模型建立的过程就是不断趋近样本函数的过程。例如真实样本函数是直线  $y=x$ ，显然我们通过线性函数可以完全拟合，如果我们采用非线性函数  $y=\sin(x)$ ，也是可以拟合的。这里有泰勒公式做支撑  $y=\sin(x)$ ，在约束范围内， $y=\sin(x)$ 的高次方趋于零。其次，激活函数 sigmoid、tanh 在 0 附近一个很小的范围内是线性的（当然，我们不希望使用这一部分），如果恰好在这一部分激活，激活函数可以看作是线性层。最后，现实中的序列很少只存在线性关系或者只存在非线性关系，如果可以使用线性模型搞定的事情，我们当然不希望使用像 LSTM 这样复杂的模型。



5.2 初始化与正则

混沌原理告诉我们，对于系统初始态任何一个微小的改变，都将导致系统朝着一个不可知的方向发展。在深度学习中，我们通常采用随机初始化来定义权重矩阵，即使在本章中采用截断高斯初始化，每一次初始化的参数也不会相同，这导致每一次训练的模型也会有差别。在多次运行的过程中，确实有极低的概率导致模型最后不收敛。正则是在初始化的基础上，引导模型往好的方向发展，这里通过梯度裁剪、dropout、学习率、提前终止四种方式对模型做了约束。其中学习率、梯度裁剪控制模型的变化程度，如果发现在训练的过程中 loss 跳变比较严重，则需要减小变化量。dropout 用于提升模型的泛化能力。提前终止是为了找到拟合与泛化最佳的平衡点。