Tensorflow_RNN

<Excerpt in index | 首页摘要>
使用Tensorflow搭建RNN,熟悉一些经典的应用。
Recurrent Neural Network　　LSTM　　GRU　　自然语言处理
Github：Tensorflow
Resource:深度学习理论与实战（基于TensorFlow实现）

<The rest of contents | 余下全文>

RNN（循环神经网络）

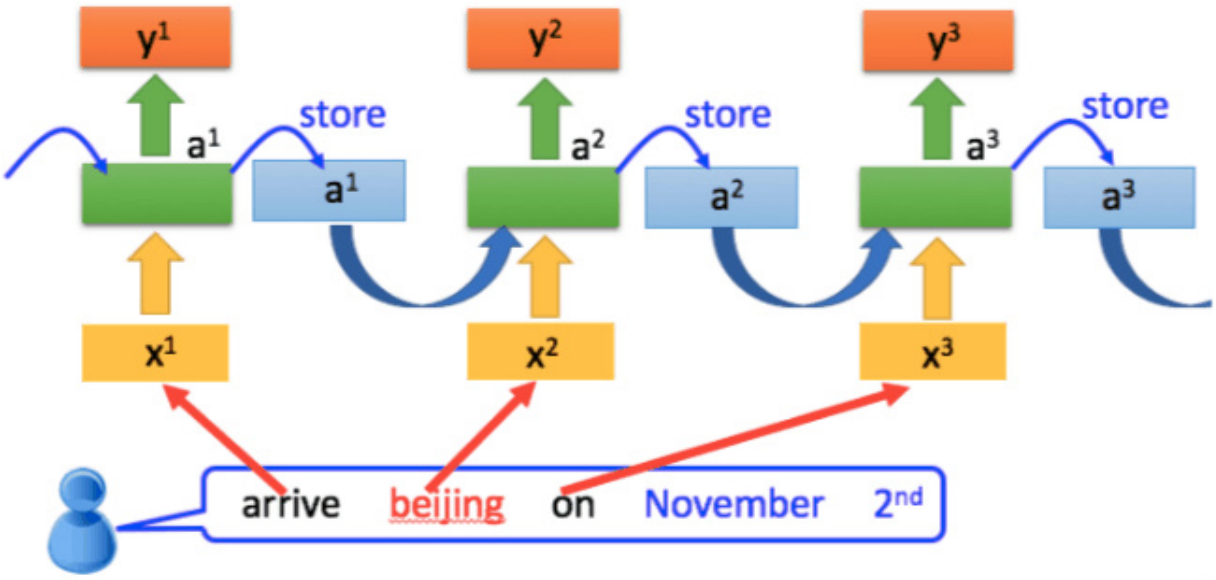
1、RNN基本认识

基本概念

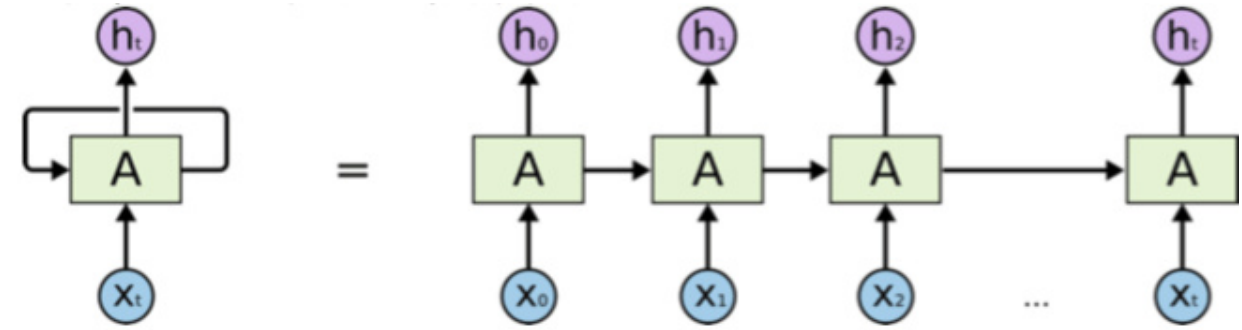
在深度学习的领域，除了视觉的问题，还有很多关于**文本，语音**的问题，这些问题不同于视觉这种结构性的问题，而是需要**时间依赖和记忆性**，这个时候使用卷积神经网络就不再那么适合，所以需要使用一种新的形式的网络，**循环神经网络。**

具有记忆效果的网络-循环神经网络

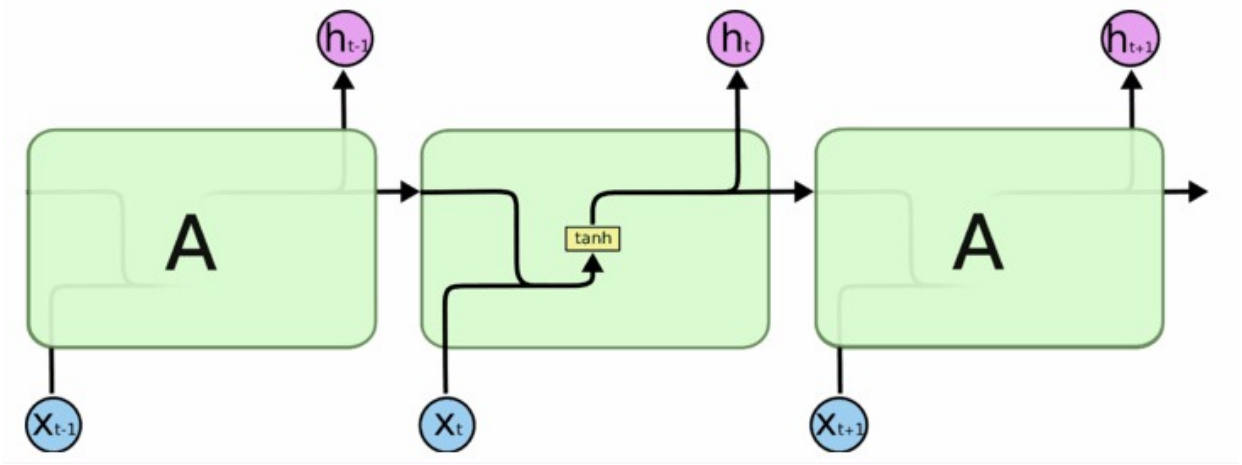
一般我们讲记忆性，都是对一连串发生的事情来讲，所以循环神经网络一般是使用在**一个序列上的**,下面对于整个序列，有如下的示意图



这里用了**参数共享**，也就是说上面虽然有 3 个绿色的格子，但是实际上它们都是同一个绿色的格子，我们可以用下面的图来形象的表示:



代码实现



对于最简单的 RNN，我们可以使用下面两种方式去调用，分别是 `tf.nn.rnn_cell.BasicRNNCell()` 和 `tf.nn.dynamic_rnn`，这两种方式的区别在于 `BasicRNNCell()` 只能接受序列中**单步的输入**，且必须传入隐藏状态，而 `dynamic_rnn()` 可以接受**一个序列的输入**，默认会传入全 0 的隐藏状态，也可以自己申明隐藏状态传入。

- `BasicRNNCell()` 里面的参数有

`num_units`: 输出的特征维度

`activation`：选用的激活函数, 默认 `tanh`

`reuse:` 是否需要复用

- `dynamic_rnn` 里面的参数则要丰富一些, 最重要的参数是下面的这些:

`inputs`: 基础的 `RNNCell`

`initial_state`: 设置初始状态

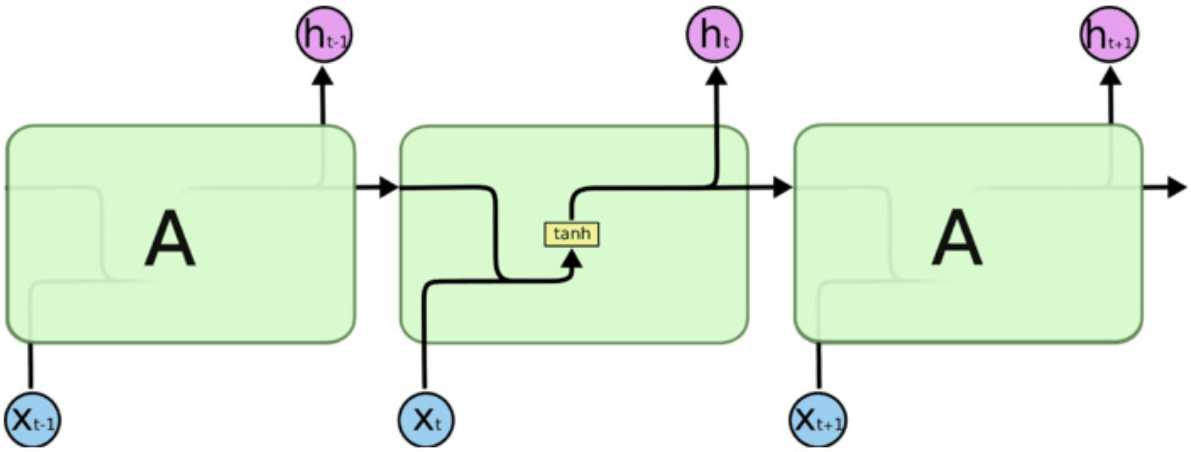
time_major: 确定时间步长是否在输入的第一维上, 如果是, 那么输入就是 `[max_time, batch_size, depth]` 的格式, 否则是 `[batch_size, max_step, depth]`

2、LSTM

前面我们讲到循环神经网络存在的长时依赖问题，后面出现了两种循环神经网络的变式 `LSTM` 和 `GRU`，这两种模型随后变得非常流行，基本现在指的**循环神经网络都是说的这两种网络**，首先我

们介绍一个最基本的 RNN。

标准RNN



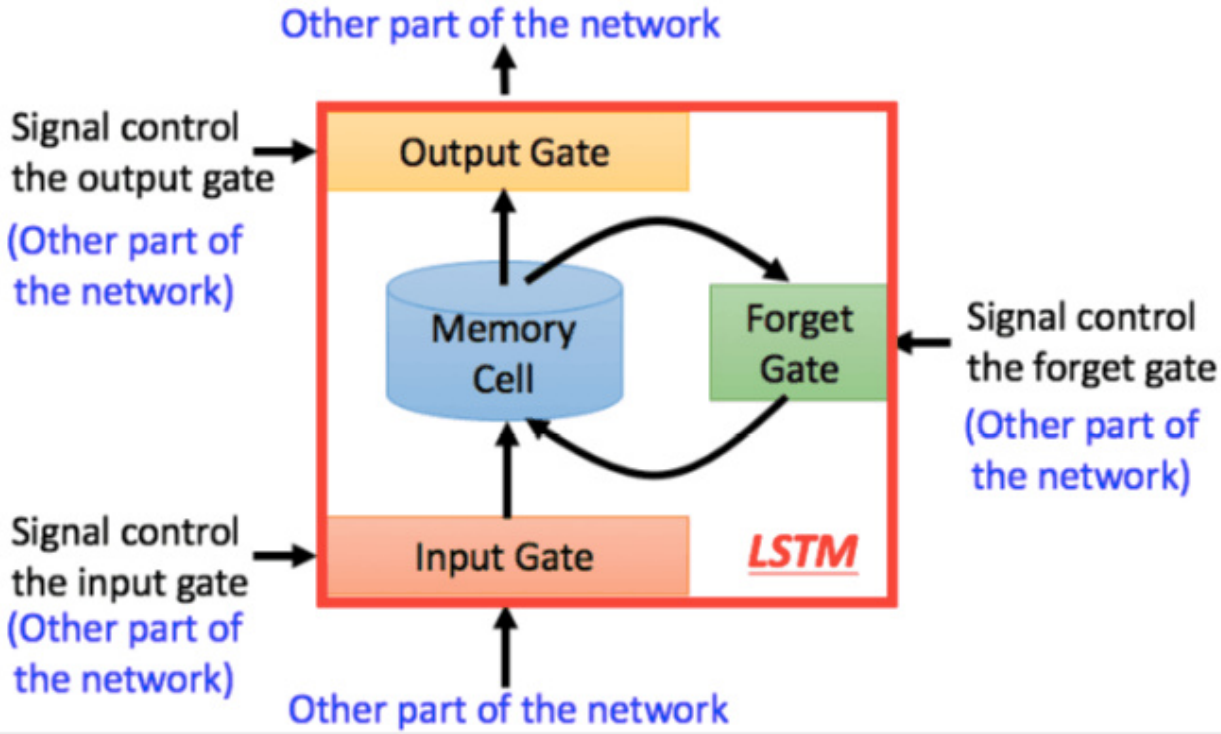
标准的 **RNN** 使用下面的公式进行计算:

$$h_t = tanh(w_{ih}x_t + b_{ih} + w_{hh}h_{t-1} + b_{hn})$$

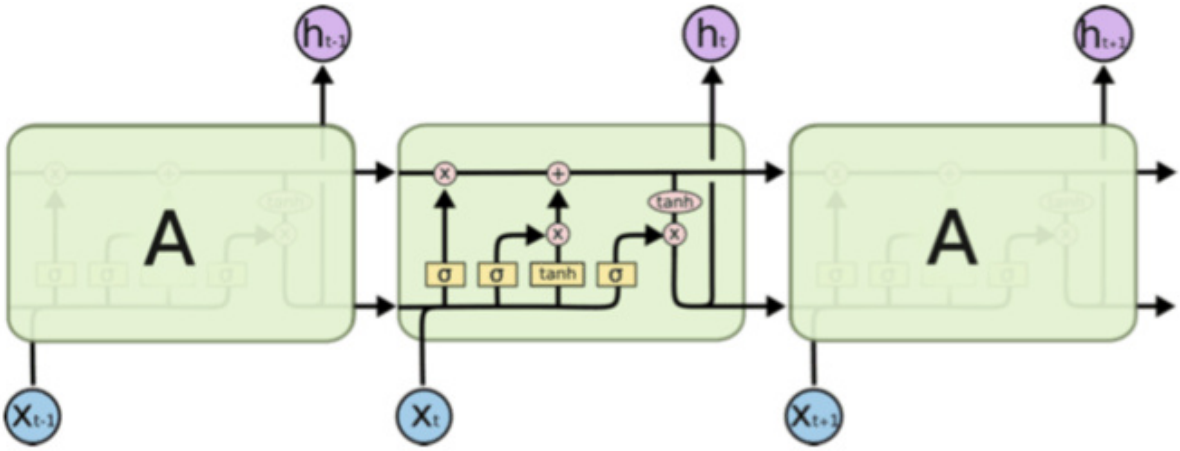
其中 h_{t-1} 表示前一步的记忆状态，x_t 表示当前步的输入，最后得到输出 h_t，同时将其传入的后面作为这一步的记忆状态。

LSTM

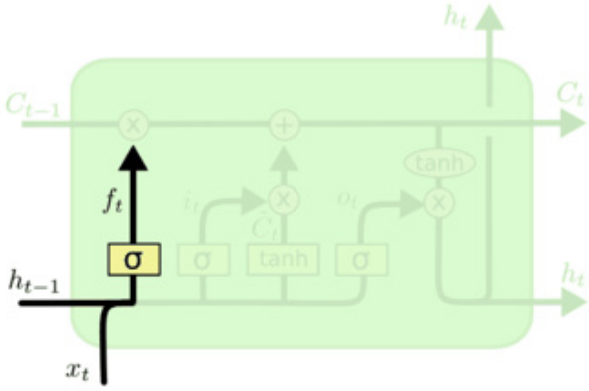
LSTM 是 Long Short Term Memory networks 的缩写，能够在一定程度上解决**长时依赖的问题。**网络的结构和之前讲的 RNN 是一样的，但是内部有着更加复杂的结构，对于**单步的抽象网络**，我们有下面的图示



LSTM 由三个门来控制，分别是**输入门，遗忘门和输出门。**输入门和输出门限制着输入和输出的大小，而遗忘门控制着记忆的保留读，对于一个任务，遗忘门能够自己学习到该保留多少以前的记忆，不需要人为进行干扰，所以**具备着长时记忆**的功能.

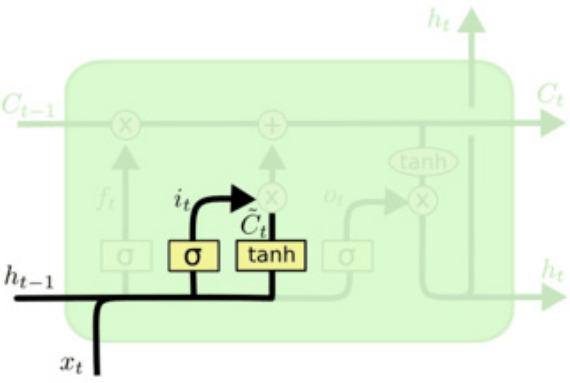


这里是三个序列的 **LSTM** 结构，每个 A 中的参数都是相同的。



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

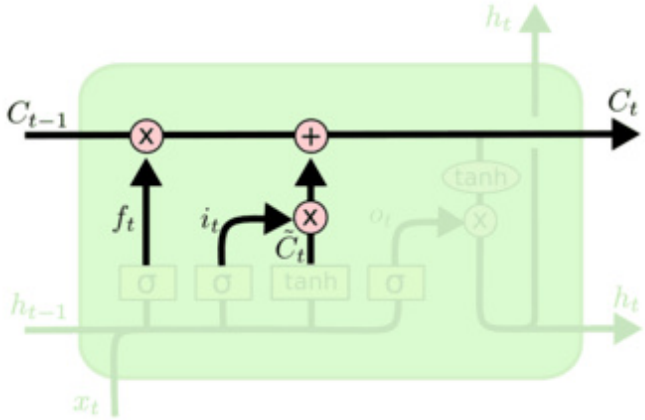
首先我们结合 t-1 时刻的网络输出和 t 时刻的网络输入，通过线性变换和 sigmoid 函数得到一个 0 ~ 1 之间的值 f_t，这个值可以称为衰减系数，表示保留过去信息的多少。



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

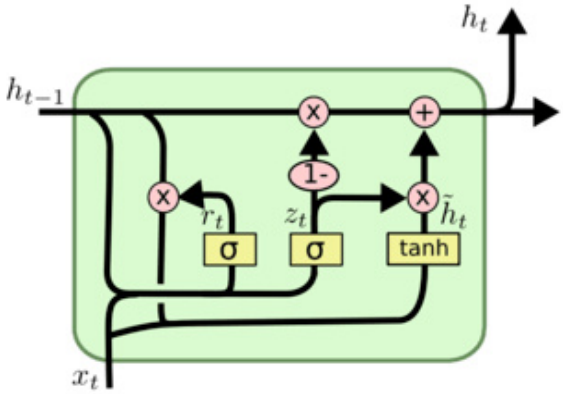
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

然后还是结合 t-1 时刻的输出和 t 时刻的输入分别计算出另外一个系数 i_t 和新接受的信息 $\tilde{C}_t$，其中 i_t 表示保留接受的新信息的多少。



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

然后将前面得到的**两个衰减系数分别乘上过去的信息和现在的新信息**来确定 t 时刻真正的记忆状态 C_t



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

最后还是使用 **t-1 时刻的输出**和 **t 时刻的输入**计算一个系数，这个系数表示 t 时刻到底输出多少的记忆状态 C_t 得到真正的模型输出 h_t.

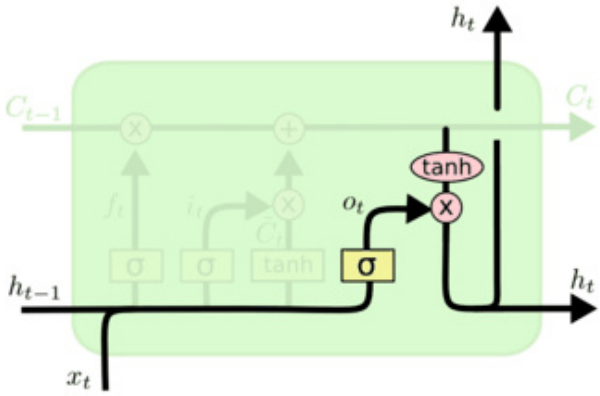
`LSTM` 和基本的 `RNN` 是一样的，他的参数也是相同，我们就不再赘述了

在 `RNN` 中, 我们也可以定义深层网络, 就是通过 `tf.nn.rnn_cell.MultiRNNCell` 来实现, 调用它的方式非常简单, 构造一个 `cell` 的 `list` 作为参数传入就可以了

- `LSTM` 和 `MultiRNNCell`

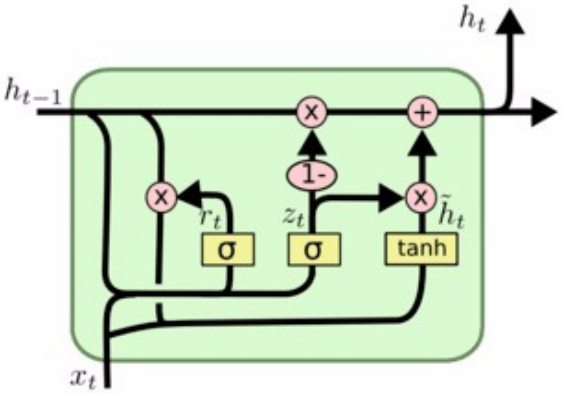
3、GRU

下面我们该说说 **GRU** 了，**GRU** 是 Gated Recurrent Unit 的缩写，由 2014 年 Cho 提出，GRU 和 LSTM 最大的不同在于 GRU 将**遗忘门和输入门合成了一个更新门**，同时网络不再额外给出记忆状态 C_t，而是将输出结果作为记忆状态不断往后传，下面是其结构图



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

本质上跟前面的 LSTM 是相同的



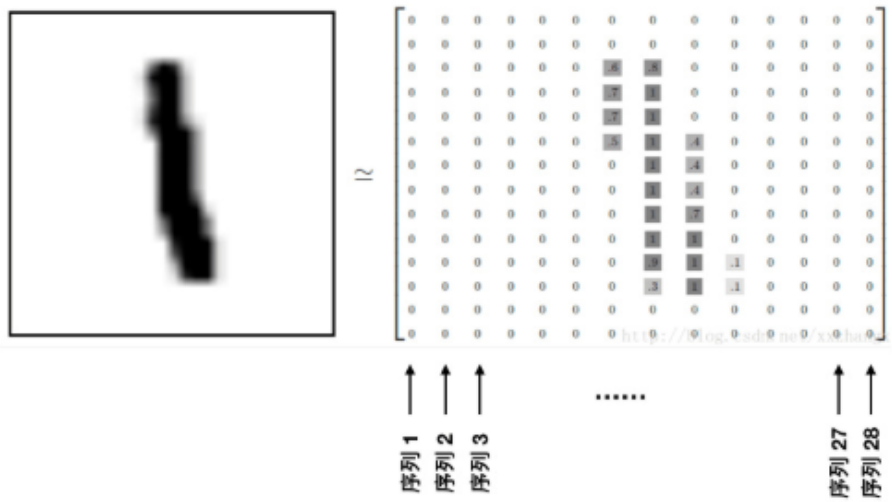
$$z_t = \sigma\left(W_z \cdot \left[h_{t-1}, x_t\right]\right)$$
$$r_t = \sigma\left(W_r \cdot \left[h_{t-1}, x_t\right]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot \left[r_t * h_{t-1}, x_t\right]\right)$$
$$h_t = \left(1 - z_t\right) * h_{t-1} + z_t * \tilde{h}_t$$

RNN的应用

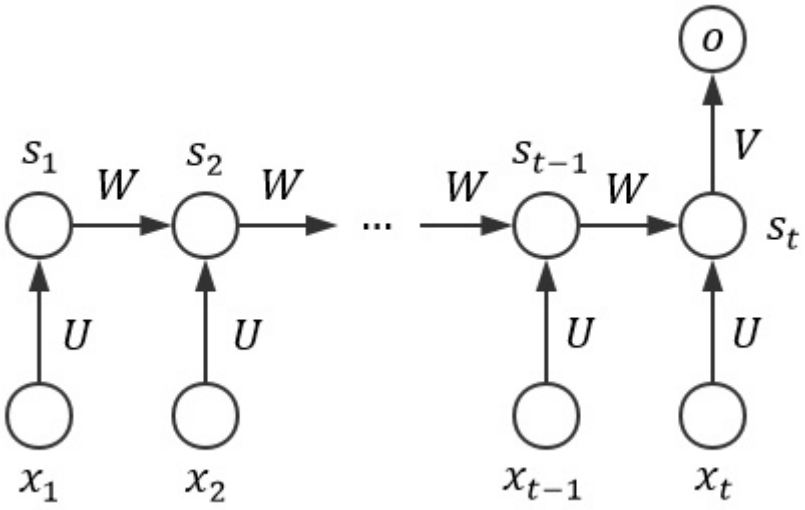
`RNN` 比较擅长处理 `时间序列关系` 的数据。

1、做图像分类

对于一张手写字体的图片，其大小是 `28 * 28`，我们可以将其看做是一个长为 28 的序列，每个序列的特征都是 28，也就是



这样我们解决了输入序列的问题，最后我们保**留最后一个结果作为输出**:



代码实现

构建单个lstm网络

```
# 初始化初始状态和cell
# LSTM
# num_units: int, The number of units in the LSTM cell.
def build_lstm(num_units, num_layers, batch_size, keep_prob=1):
    def build_cell(num_units, keep_prob):
        cell = tf.nn.rnn_cell.LSTMCell(num_units)   #Initialize the param
eters for an LSTM cell.
        cell = tf.nn.rnn_cell.DropoutWrapper(cell, output_keep_prob=keep_
prob)  #Operator adding dropout to inputs and outputs of the given cell.
        return cell
    cell = tf.nn.rnn_cell.MultiRNNCell([build_cell(num_units, keep_prob)
for _ in range(num_layers)])      #生成多层RNN网络
    init_state = cell.zero_state(batch_size, tf.float32)         #初始状态
    return cell, init_state
```

构建整个lstm网络，得到输出

```
#    x:输入
#    num_units: 中间隐藏层的大小
#    num_layers: 隐藏层的数量
#    keep_prob: float between 0 and 1, output keep probability; if it is c
onstant and 1, no output dropout will be added.
def lstm(x, num_units, num_layers, batch_size, init_state=None, keep_prob
=1, time_major=True, scope='lstm', reuse=None):
    with tf.variable_scope(scope, reuse=reuse):
        cell, zero_state = build_lstm(num_units, num_layers, batch_size,
 keep_prob)
        if init_state is not None:
```

```
            out, final_state = tf.nn.dynamic_rnn(cell, x, initial_state=i
nit_state, time_major=time_major)
        else:
            out, final_state = tf.nn.dynamic_rnn(cell, x, initial_state=z
ero_state, time_major=time_major)
    return out, final_state
```

、

将数据转为RNN输入模式

```
inputs = tf.transpose(tf.squeeze(input_ph,axis=[-1]),(1 , 0, 2))
#删除最后一维度然后将第二维度和第一维度换一下
```

2、自然语言处理

词嵌入

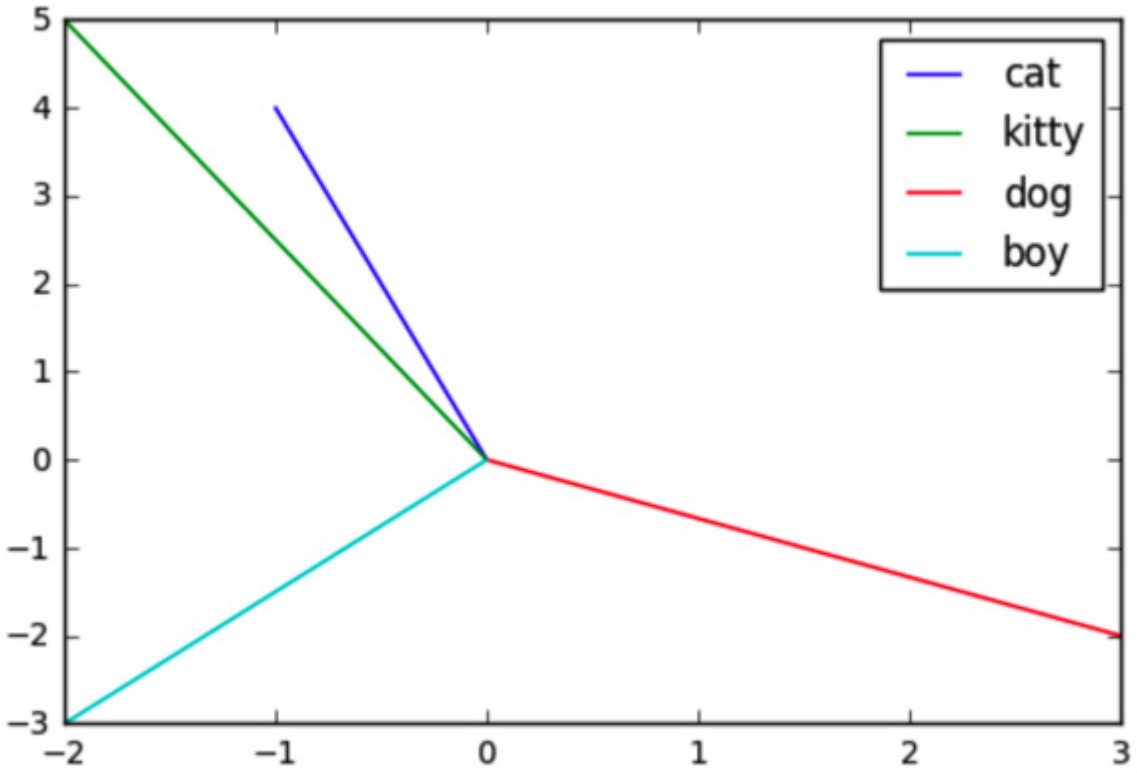
词向量简单来说就是用**一个向量去表示一个词语**，我们需要对每个词有一个特定的向量去表示他们，而有一些词的词性是相近的，比如”(love)喜欢”和”(like)爱”，对于这种词性相近的词，我们需要他们的向量表示也能够相近，如何去度量和定义向量之间的相近呢？非常简单，就是使用**两个向量的夹角，夹角越小，越相近**，这样就有了一个完备的定义。

我们举一个例子，下面有 4 段话

- The cat likes playing wool.
- The kitty likes playing wool.
- The dog likes playing ball.
- The boy likes playing ball.

这里面有 4 个词，分别是 cat, kitty, dog 和 boy。下面我们使用一个`二维的词向量（a，b）`来表示每一个词，其中 a，b 分别代表着这个词的一种`属性`，比如`a 代表是否喜欢玩飞盘`，`b 代表是否喜欢玩毛线`，数值越大表示越喜欢，那么我们就能够用数值来定义每一个单词。

对于 **cat**，我们可以定义它的词嵌入为 (-1, 4)，因为他不喜欢玩飞盘，喜欢玩毛线，其他如下：



代码实现：词嵌入

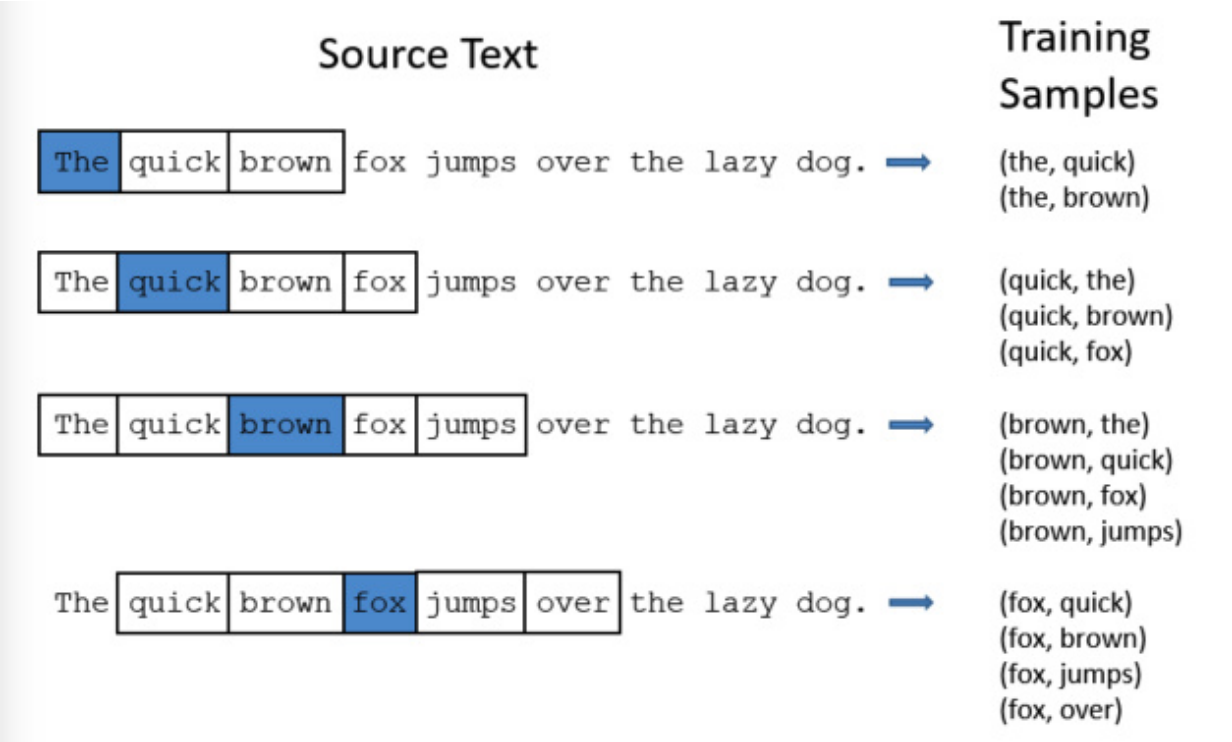
Skip-Gram 模型

Skip Gram 模型是 Word2Vec 论文的网络架构，下面我们来讲一讲这个模型。

`skip-gram` 模型非常简单，我们在一段文本中训练一个简单的网络，这个网络的任务是通过 `一个词周围的词` 来预测这个词，然而我们实际上要做的就是训练我们的词嵌入。

比如我们给定一句话中的一个词，看看它周围的词，然后随机挑选一个，我们希望网络能够输出一个 `概率值`，这个概率值能够告诉我们到底这个词离我们 `选择的词的远近程度`，比如这么一句话 **‘A dog is playing with a ball’**，如果我们选的词是 ‘**ball**‘，那么 ‘**playing**’ 就要比 ‘**dog**’ 离我们选择的词更近。

对于一段话，我们可以 `按照顺序选择不同的词`，`然后构建训练样本和 label`，比如



对于这个例子，我们依次取一个词以及其周围的词构成一个训练样本，比如第一次选择的词是 `'the'`，那么我们取其前后两个词作为 `训练样本`，这个也可以被称为一个滑动窗口，对于第一个词，其左边没有单词，所以训练集就是三个词，然后我们在这三个词中 `选择 'the' 作为输入，另外两个词都是他的输出`，就构成了 `两个训练样本`，又比如选择 'fox' 这个词，那么加上其左边两个词，右边两个词，一共是 5 个词，然后选择 'fox' 作为输入，那么输出就是其周围的四个词，一共可以构成 4 个训练样本，通过这个办法，我们就能够训练出需要的词嵌入。

N-Gram 模型

对于一句话，单词的排列顺序是非常重要的，所以我们能否由前面的几个词来预测后面的几个单词呢，比如 `'I lived in France for 10 years, I can speak _'` 这句话中，我们能够预测出最后一个词是 `French`。

对于一句话 T，其由 $w_1, w_2, \cdots, w_n$ 这 n 个词构成，

$$P(T) = P(w_1)P(w_2|w_1)P(w_3|w_2w_1)\cdots P(w_n|w_{n-1}w_{n-2}\cdots w_2w_1)$$

我们可以再简化一下这个模型，比如对于一个词，它并不需要前面所有的词作为条件概率，也就是说一个词可以只与其前面的几个词有关，这就是 `马尔科夫假设`。

对于这里的条件概率，传统的方法是统计语料中每个词出现的频率，根据**贝叶斯定理**来估计这个条件概率，这里我们就可以用**词嵌入对其进行代替**，然后使用 RNN 进行**条件概率的计算**，然后最大化这个条件概率不仅修改词嵌入，同时能够使得模型可以依据计算的条件概率对其中的一个单词进行预测。

代码实现：由前面的几个词来预测后面的几个单词

将单词三个分组，前面两个作为输入，最后一个作为预测的结果。

定义模型

```
def n_gram(inputs, vocab_size, context_size=CONTEXT_SIZE, n_dim=EMBEDDING
_DIM, scope='n-gram', reuse=None):
    with tf.variable_scope(scope, reuse=reuse):
        with tf.device('/cpu:0'):
            embeddings = tf.get_variable('embeddings', shape=[vocab_size,
 n_dim], initializer=tf.random_uniform_initializer)
        embed = tf.nn.embedding_lookup(embeddings, inputs)

        net = tf.reshape(embed, (1, -1))
        net = slim.fully_connected(net, vocab_size, activation_fn=None, s
cope='classification')

        return net
```

定义优化器和loss

```
loss = tf.losses.sparse_softmax_cross_entropy(label_ph, net,
scope='loss')
opt = tf.train.MomentumOptimizer(1e-2, 0.9)
train_op = opt.minimize(loss)
```

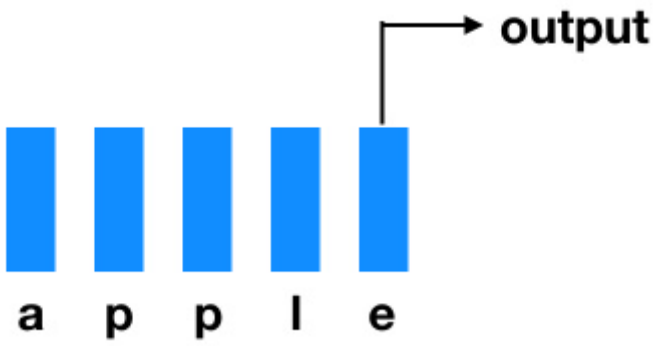
开始训练

```
for e in range(100):
    train_loss = 0
    for word, label in trigram[:100]:
        word = [word_to_idx[i] for i in word]
        label = [word_to_idx[label]]

        _, curr_loss = sess.run([train_op, loss], feed_dict={input_ph: wo
rd, label_ph: label})
        train_loss += curr_loss

    if (e + 1) % 20 == 0:
        print('Epoch: {}, Loss: {:.6f}'.format(e + 1, train_loss / 100))
```

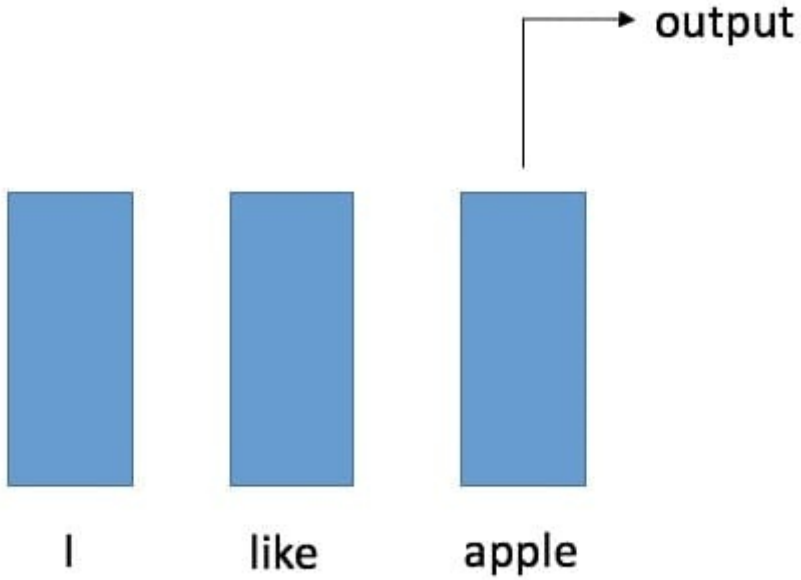
LSTM 做词性预测

对于一个单词，会有这不同的词性，首先能够根据一个单词的后缀来初步判断，比如`-ly`这种后缀，很大概率是一个`副词`，除此之外，一个相同的单词可以表示两种不同的词性，比如 book 既可以表示名词，也可以表示动词，所以到底这个词是什么词性需要`结合前后文来具体判断`。

根据这个问题，我们可以使用`lstm 模型`来进行预测，首先对于一个单词，可以将其看作一个序列，比如`apple`是由 a p p l e 这 5 个单词构成，这就形成了长度为 5 的序列，我们可以对这些字符构建词嵌入，然后输入 lstm，就像 lstm 做图像分类一样，只取`最后一个输出作为预测结果`，整个单词的字符串能够形成一种`记忆的特性`，帮助我们更好的预测词性。



接着我们把这个单词和其前面几个单词**构成序列**，可以对这些单词构建新的词嵌入，最后输出结果是**单词的词性**，也就是根据前面几个词的信息对这个**词的词性进行分类**。



代码实现：LSTM 做词性预测

构建词性分类模型

```
def lstm_tagger(word_code, word_list, n_word, n_char, word_dim, char_dim,


          word_hidden, char_hidden, n_tag, scope='lstm_tagger', reuse=No
ne):
with tf.variable_scope(scope, reuse=reuse):
    # 首先对一个句子里的所有单词用`char_lstm`进行编码
    def char_lstm_fun(single_word):
        # 使用`tf.string_split`对单词进行字母级别的分割
        char_list = tf.string_split([single_word], delimiter='').values

        # 使用`char_table`查找所有字母的编码
        char_code = char_table.lookup(char_list)

        # 将编码进入`lstm`得到输出
        char_lstm_out = char_lstm(char_code, len(chars), 10, char_hidden)
```

```
        return char_lstm_out

    # tf.while_loop退出循环的条件
    def cond(i, char_out_list, word_list):
        return tf.less(i, tf.shape(word_list)[0])

    # tf.while_loop循环体
    def body(i, char_out_list, word_list):
        char_out = char_lstm_fun(word_list[i])
        # 如果不是第一个，在第1维进行 concate，否则直接赋值
        char_out_list = tf.cond(i > 0, lambda: tf.concat([char_out_list,
 char_out], axis=0), lambda: char_out)

        return i + 1, char_out_list, word_list

    # tf.while_loop的初始变量 i
    i = tf.constant(0)

    # tf.while_loop的初始变量 init_char_list
    ## 为了更少debug，用一个形状为 (None, char_hidden) 的 placeholder 作为初始
化，这样
    ## 经过 tf.while_loop 之后形状不会发生改变
    ## 但这增加了一个实际上没有用的 placeholder，或许有更优的解决方法
    init_char_list = tf.placeholder(tf.float32, shape=(None, char_hidde
n))

    # tf.while_loop
    # 三个参数，第一个是退出循环条件函数，第二个是循环体函数，第三个是带入的初始变量
    # 可以参考 https://blog.csdn.net/qq_20611245/article/details/77363609
    _, char_out_list, _ = tf.while_loop(cond, body, [i, init_char_list, w
ord_list])

    # 最后将形状从 (seq, char_hidden) --> (seq, 1, char_hidden)
    char_out = tf.expand_dims(char_out_list, axis=1)

    # 构造单词的嵌入模型
    word_embeddings = tf.get_variable('embeddings', shape=(n_word, word_d
im),
                                      dtype=tf.float32, initializer=tf.ra
ndom_uniform_initializer(minval=0.0, maxval=1.0))
    word_embed = tf.nn.embedding_lookup(word_embeddings, word_code, name
='word_embed')# (seq, word_dim)
    word_embed = tf.expand_dims(word_embed, axis=1) # (seq, 1, word_dim)

    # 将单词的嵌入向量和单词的`lstm`结果按照最后一维(特征)进行连接
    net = tf.concat([char_out, word_embed], axis=-1)

    # 进入`lstm`
    net, _ = lstm(net, word_hidden, 1, 1)

    # 分类层
```

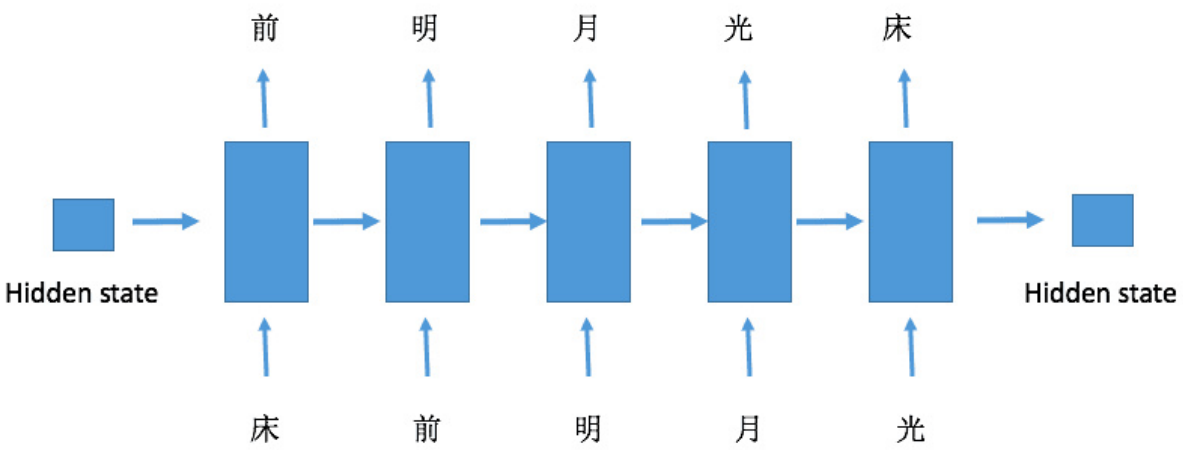
```
    net = tf.reshape(net, (-1, word_hidden))
    net = slim.fully_connected(net, n_tag, activation_fn=None, scope='cla
ssification')

    return net, init_char_list
```

用RNN生成古诗

原理介绍

前面我们介绍过 RNN 的输入和输出存在多种关系，比如多个输入对一个输出，这个时候输入是一个序列，输出是一个分类结果，就像使用 RNN 做图像分类。

这里我们使用 RNN 来生成文本，网络的输入是**一个序列**，同时输出也是一个**相同长度的序列**，结构如下



具体实现以后再补充

反馈与建议

- 微博：@柏林designer
- 邮箱：wwj123@zju.edu.cn