**AI**

**实 验 报 告**

**学生姓名** 吴语港

**学生学号** SA19225404

**实验日期** 2019/10/30

**实验名称** 基于RNN-LSTM+CTC

的注册码识别实践

**学生学号** SA19225404

**实验地点** 思贤楼301 303实验室

1. **实验目的:**

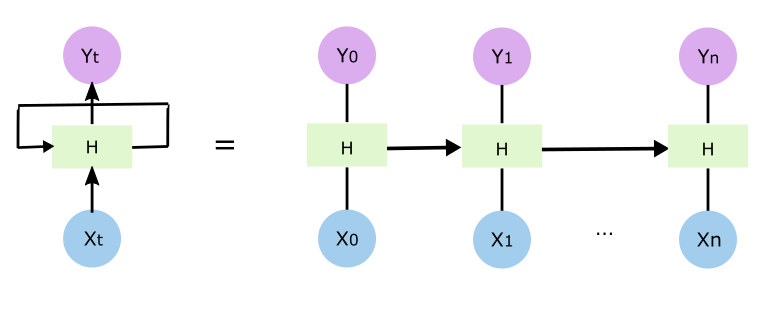
通过CNN+LSTM+CTC实现变长字符串的识别

1. **实验原理:**

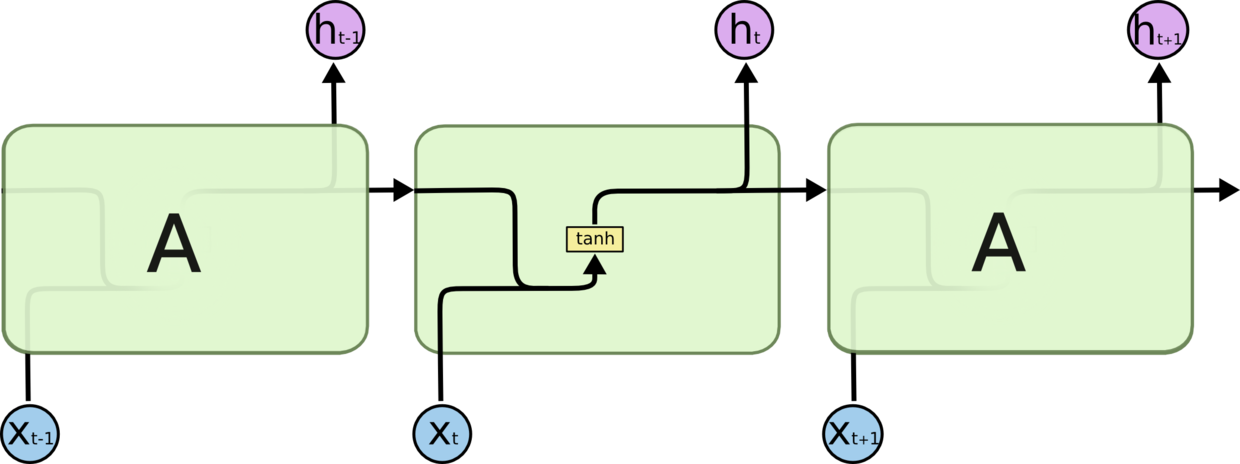
**（1）循环神经网络RNN(Recurrent neural network)**

递归神经网络是两种人工神经网络的总称：时间递归神经网络（recurrent neural network）和结构递归神经网络（recursive neural network）。时间递归神经网络的神经元间连接构成有向图，而结构递归神经网络利用相似的神经网络结构递归构造更为复杂的深度网络。

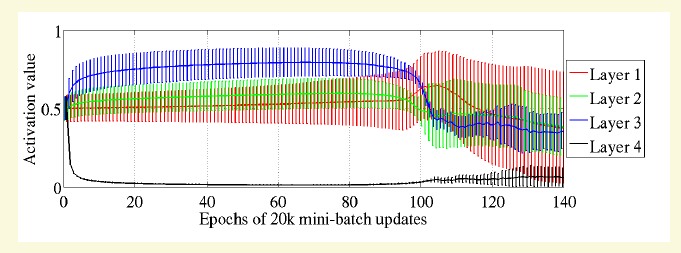
我们常说的RNN一般指代时间递归神经网络。单纯递归神经网络因为无法处理随着递归，权重指数级爆炸或消失的问题（Vanishing/Exploding gradient problem），难以捕捉长期时间关联；而结合不同的LSTM可以很好解决这个问题。时间递归神经网络可以描述动态时间行为，因为和前馈神经网络（feedforward neural network）接受较特定结构的输入不同，RNN将状态在自身网络中循环传递，因此可以接受更广泛的时间序列结构输入。手写识别是最早成功利用RNN的研究结果。

****

两个输出结果：一是当前层输出结果，二是参与下一层运算输出中间结果

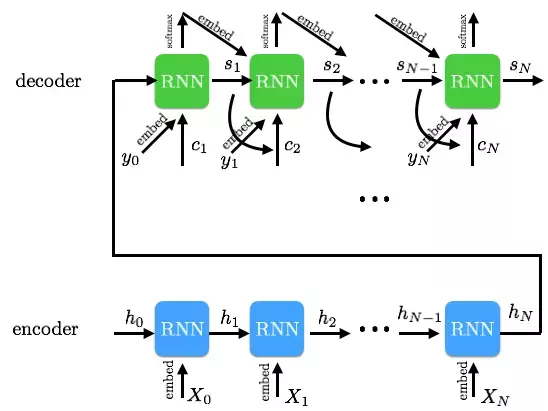
****

但是，RNN在处理长期依赖（时间序列上距离较远的节点）时，距离较远的节点之间的联系时会涉及雅可比矩阵的多次相乘，这会带来梯度消失（经常发生）或者梯度膨胀（较少发生）的问题

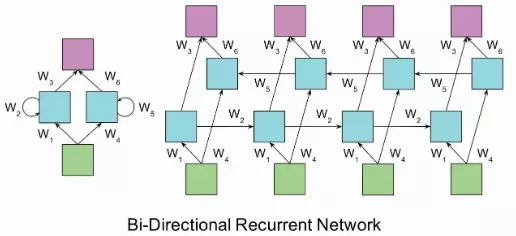


分类：

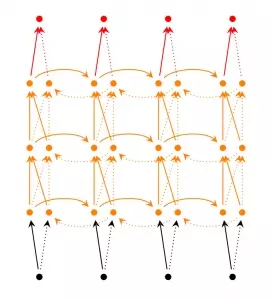
编码器-解码器



双向RNN

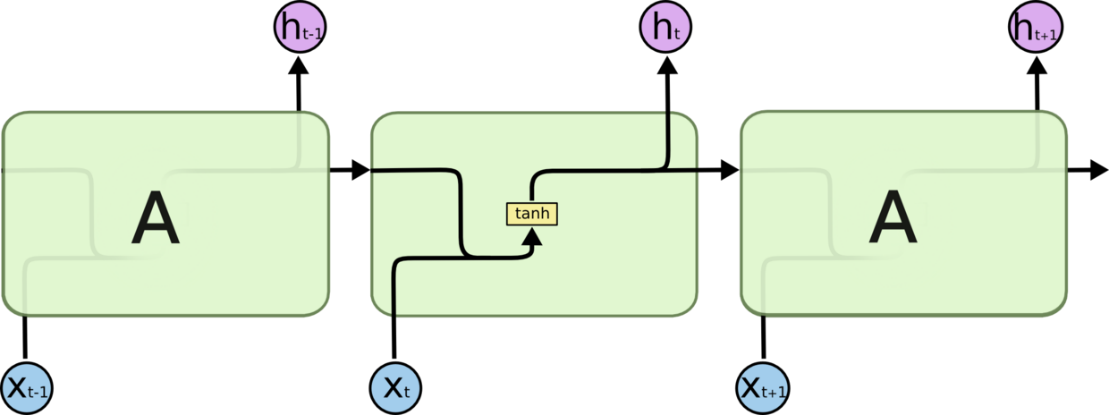


深层双向RNN

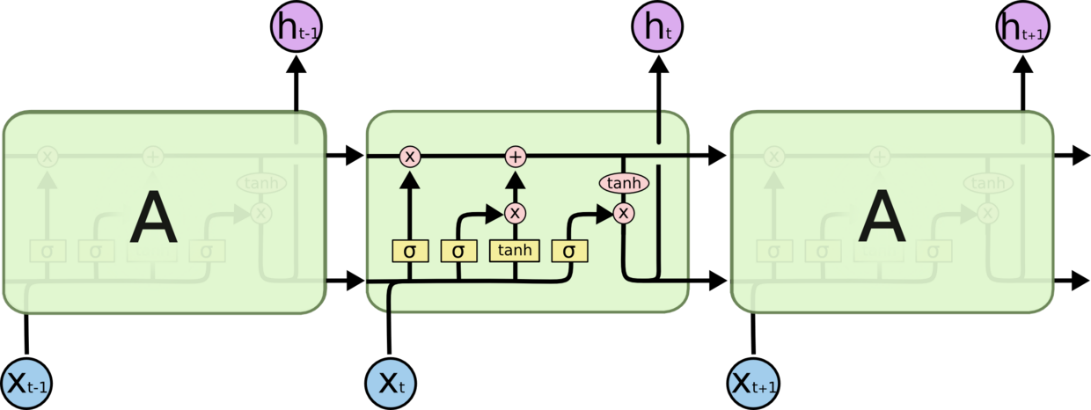


**（2）长短期记忆网络 LSTM（Long short-term memory）**

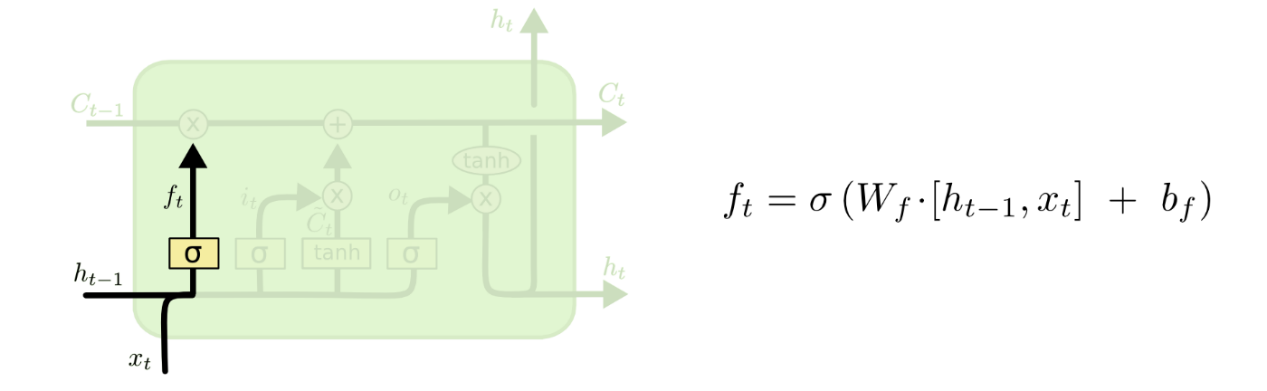
LSTM算法全称为Long short-term memory，最早由 Sepp Hochreiter和Jürgen Schmidhuber于1997年提出，是一种特定形式的RNN，可以解决传统RNN的长期依赖(long-term dependencies)问题。它对传统RNN的隐层进行了结构上的改进。

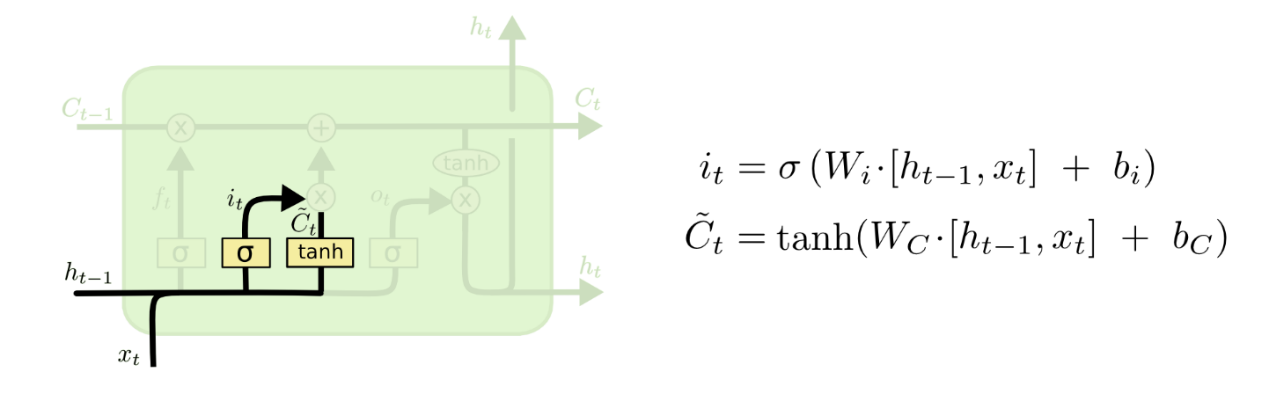


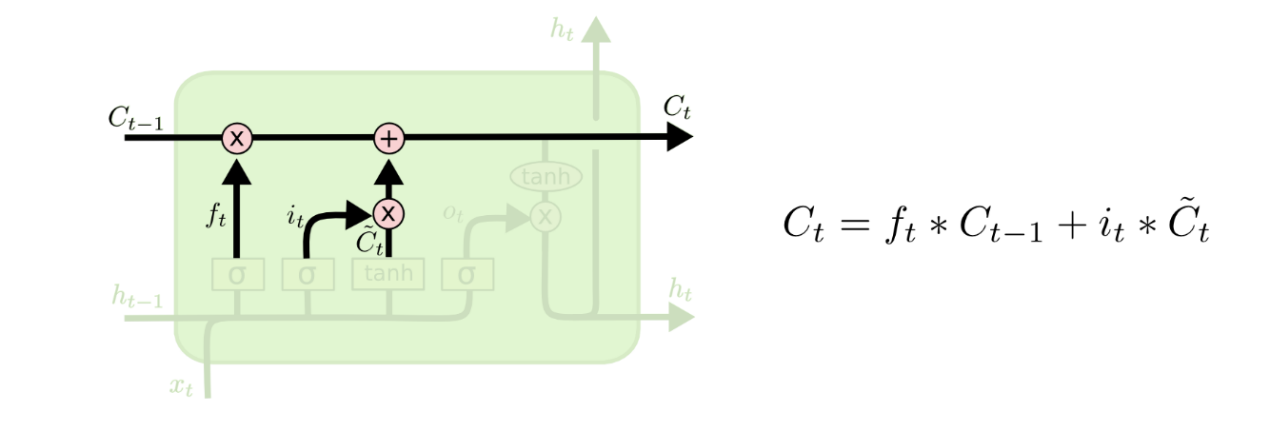
这是传统的RNN的结构，内部有一个tanh层

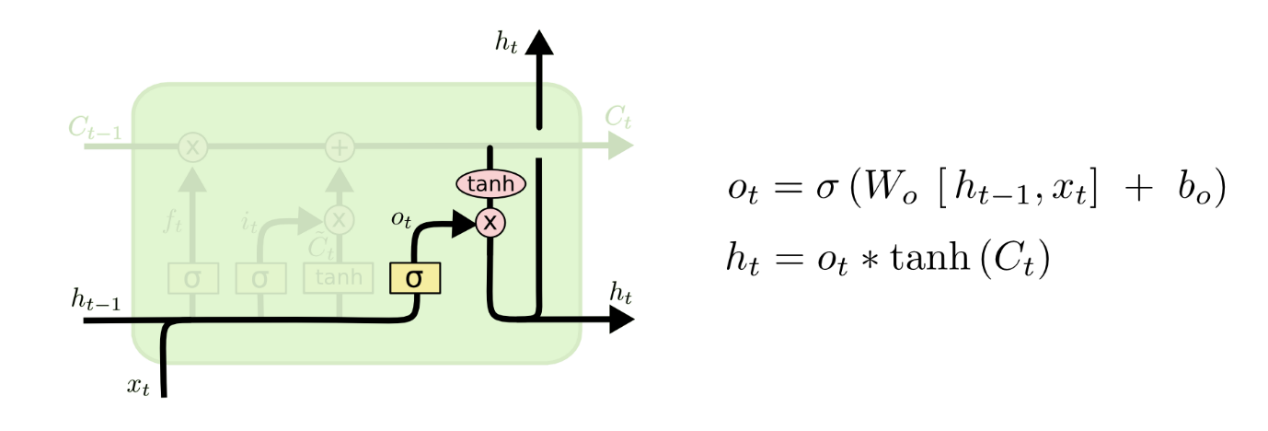


LSTM和传统RNN结构类似，然而内部结构却有所不同

LSTM中有三种门：遗忘门：  
  
可以看到这里的ft由输入的xt和ht−1得到，用来控制Ct−1中的信息的遗忘程度。ft中的每个值都是0-1中的一个数，下界0代表完全遗忘，上界1代表完全不变。

输入门：  
  
遗忘门决定了历史状态信息的遗忘程度，那么输入门的作用就是往状态信息中添加新东西。同样，由输入的xt和ht−1得到当前的it用以控制新状态信息的更新程度。这里新状态信息C~也是通过输入的xt和ht−1计算得出。

  
那么当前新的状态信息Ct就很显然可以通过上式计算得出，通俗的说就是遗忘一些旧信息，更新一些新信息进去。

输出门：  
  
最后就是输出门了。类似地，根据xt和ht−1得出ot用以控制哪些信息需要作为输出。

概括一下：

状态信息Ct的依赖于遗忘门ft和输入门it

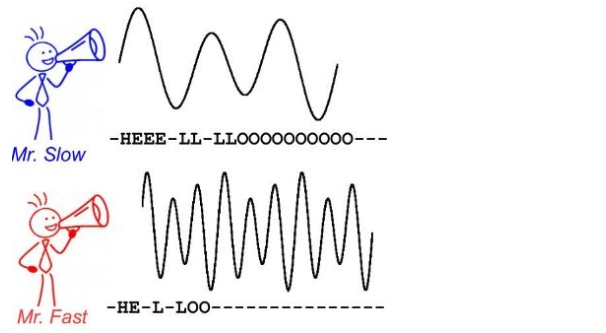
遗忘门ft和输入门it依赖于输入参数中的ht−1

而当前隐层输出ht依赖于Ct

**（3）联结主义时间分类器 CTC（Connectionist Temporal Classifier）**

CTC，Connectionist Temporal Classification，用来解决输入序列和输出序列难以一一对应的问题。

举例来说，在语音识别中，我们希望音频中的音素和翻译后的字符可以一一对应，这是训练时一个很天然的想法。但是要对齐是一件很困难的事，如下图所示，有人说话块，有人说话慢，每个人说话快慢不同，不可能手动地对音素和字符对齐，这样太耗时。



再比如，在OCR中使用RNN时，RNN的每一个输出要对应到字符图像中的每一个位置，要手工做这样的标记工作量太大，而且图像中的字符数量不同，字体样式不同，大小不同，导致输出不一定能和每个字符一一对应。

****

**CTC的特征：**

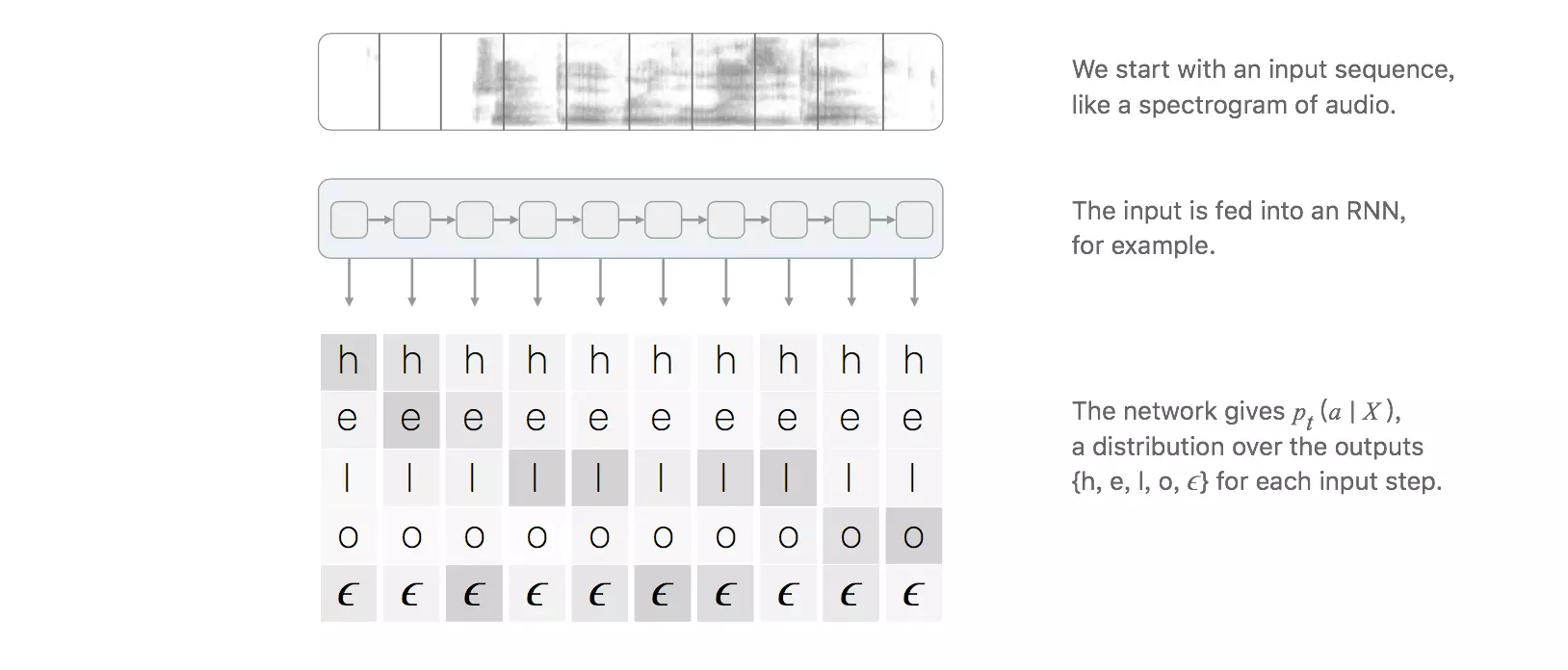
条件独立：CTC的一个非常不合理的假设是其假设每个时间片都是相互独立的，这是一个非常不好的假设。在OCR或者语音识别中，各个时间片之间是含有一些语义信息的，所以如果能够在CTC中加入语言模型的话效果应该会有提升。

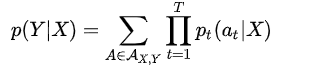
单调对齐：CTC的另外一个约束是输入与输出之间的单调对齐，在OCR和语音识别中，这种约束是成立的。但是在一些场景中例如机器翻译，这个约束便无效了。

多对一映射：CTC的又一个约束是输入序列的长度大于标签数据 的长度，但是对于的长度大于的长度的场景，CTC便失效了。

**训练（损失函数）：**

这里要明确一点，比如对于一个标定好的音频片段，训练该片段时，我们希望的输出就是标定的文本，如下图所示，音频说的一个hello，RNN或者其他模型输出的是相同数量的向量，向量里是每个字母的概率



对于一对输入输出来说，CTC的目标是将下式概率最大化  
 

解释一下，对于RNN+CTC模型来说，RNN输出的就是概率，t表示的是RNN里面的时间的概念。乘法表示一条路径的所有字符概率相乘，加法表示多条路径。因为上面说过CTC对齐输入输出是多对一的，例如对应的都是“hello”，这就是输出的其中两条路径，要将所有的路径相加才是输出的条件概率

但是对于一个输出，路径会非常的多，这样直接计算概率是不现实的，CTC算法采用动态规划的思想来求解输出的条件概率处理输出标签和真实标签之间的损失，解决**输出标签数和真实标签数不对齐**的问题，CTC\_LOSS函数如下所示：

def tf.nn.ctc\_loss(labels, inputs, sequence\_length, preprocess\_collapse\_repeated=False, ctc\_merge\_repeated=True, ignore\_longer\_outputs\_than\_inputs=False, time\_major=True):

输入参数：

labels: 是一个 int32 类型的稀疏张量（SparseTensor）， labels.indices[i, :] == [b, t] 表示 labels.values[i] 保存着(batch b, time t)的 id，labels.values[i] must take on values in [0, num\_labels)

inputs: （常用变量 logits 表示）经过 RNN 后输出的标签预测值，是一个 3D 浮点 Tensor，当 time\_major=True(默认)时形状为：(max\_time \* batch\_size \* num\_classes)，否则形状为：batch\_size \* max\_time \* num\_classes，ctc\_loss will perform the softmax operation for you

sequence\_length: 1-D int32 vector, size 为 [batch\_size]，vector 中的每个值表示序列的长度，形如 [max\_time\_step,…,max\_time\_step] ，此 sequence\_length 和用在 dynamic\_rnn 中的 sequence\_length 是一致的， 用来表示 rnn 的哪些输出不是 pad 的.

preprocess\_collapse\_repeated: 是否需要预处理，将重复的 label 合并成一个，默认是 False

ctc\_merge\_repeated: 默认为 True

输出：

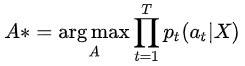
A 1-D float Tensor, size [batch], containing the negative log probabilities，同样也需要对 ctc\_loss 求均值。

**预测（解码）：**

当我们训练好一个模型后，我们输入，我们的目的是计算下式得到输出



1.一种方法是贪婪算法，取RNN每次输出概率最大的节点，计算方式如下



然后通过去重得到输出结果。

def tf.nn.ctc\_greedy\_decoder(inputs, sequence\_length, merge\_repeated=True):

输入参数：

inputs: 一个3D Tensor (max\_time \* batch\_size \* num\_classes)，保存着 logits.(通常是RNN接上一个线性神经元的输出)

sequence\_length: 1-D int32 向量, size为 [batch\_size]. 序列的长度.此 sequence\_length 和用在dynamic\_rnn 中的 sequence\_length 是一致的, 使用来表示 rnn 的哪些输出不是 pad 的.

merge\_repeated: Boolean. Default: True.

输出：

一个 tuple (decoded, neg\_sum\_logits)

decoded: 一个只有一个元素的 list，decoded[0] 是一个SparseTensor,保存着解码的结果.

decoded[0].indices: 索引矩阵，size 为(total\_decoded\_outputs \* 2)，每行中保存着[batch, time ]

decoded[0].values: 值向量，size 为 (total\_decoded\_outputs)，向量中保存的是解码的类别(decoded classes)

decoded[0].shape: Shape vector，size (2)，值为[batch\_size, max\_decoded\_length]

neg\_sum\_logits: 一个浮点型矩阵，size 为 (batch\_size\*1)，for the sequence found, the negative of the sum of the greatest logit at each timeframe

2.第二种算法是Beam search的一种变形

先来说一下Beam search算法，该算法有个参数叫做宽度，假设宽度设为3，在RNN的输出中，该算法每个时间t输出时，不同于贪婪算法只找最高的，而是找最高的三个概率作为下一次的输入，依次迭代，如下图所示，每次t时间都是基于t-1输出的最高三个查找当前概率最高的三个。(这里也可以看出，当宽度设置为1时就是贪婪算法)

def tf.nn.ctc.beam\_search\_decoder(inputs, sequence\_length, merge\_repeated=True):

输入参数：

inputs: 一个3D Tensor (max\_time \* batch\_size \* num\_classes)，保存着 logits.(通常是RNN接上一个线性神经元的输出)

sequence\_length: 1-D int32 向量, size为 [batch\_size]. 序列的长度.此 sequence\_length 和用在dynamic\_rnn 中的 sequence\_length 是一致的, 使用来表示 rnn 的哪些输出不是 pad 的.

merge\_repeated: Boolean. Default: True.

输出：

一个 tuple (decoded, neg\_sum\_logits)

decoded: 一个只有一个元素的 list，decoded[0] 是一个SparseTensor,保存着解码的结果.

decoded[0].indices: 索引矩阵，size 为(total\_decoded\_outputs \* 2)，每行中保存着[batch, time ]

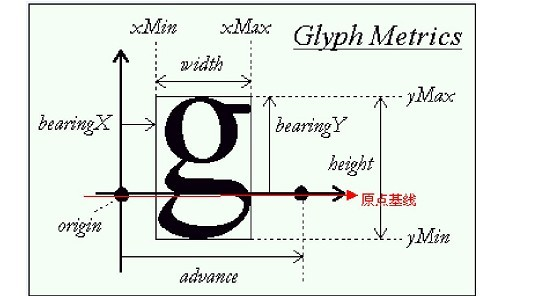
decoded[0].values: 值向量，size 为 (total\_decoded\_outputs)，向量中保存的是解码的类别(decoded classes)

decoded[0].shape: Shape vector，size (2)，值为[batch\_size, max\_decoded\_length]

neg\_sum\_logits: 一个浮点型矩阵，size 为 (batch\_size\*1)，for the sequence found, the negative of the sum of the greatest logit at each timeframe

**（4） Freetype与captcha：**

FreeType库是一个完全免费（开源）的、高质量的且可移植的字体引擎，它提供统一的接口来访问多种字体格式文件，包括TrueType, OpenType, Type1, CID, CFF, Windows FON/FNT, X11 PCF等。



CAPTCHA项目是Completely Automated Public Turing Test to Tell Computers and Humans Apart (全自动区分计算机和人类的图灵测试)的简称，卡内基梅隆大学试图将其注册为商标，但2008年请求被驳回。 CAPTCHA的目的是区分计算机和人类的一种程序算法，是一种区分用户是计算机和人的计算程序，这种程序必须能生成并评价人类能很容易通过但计算机却通不过的测试。



**（5） SparseTensor介绍：**

**1.** **SparseTensor 类介绍**

稀疏矩阵： 当密集矩阵中大部分的数都是 0 的时候，就可以用一种更好的存储方式（只将矩阵中不为 0 的）

Class tf.SparseTensor(indices, values, dense\_shape):

输入参数：

indices： 指定 Sparse Tensor 中非 0 值的索引，是一个 2D 的 int64 张量，形状为[N, ndims]，其中 N 为非 0 值的维数，ndims 为 dense\_shape 的维数

values： 指定 Sparse Tensor 索引处的值，是 一个1D 张量，维数为[N]

dense\_shape： 指定 Sparse Tensor 的形状，是一个 1D 的 int64 张量，维数为[ndims]，代表原来密集矩阵的形状

输出：

形状为 dense\_shape、指定索引 indices 处的值为 values 的稀疏张量

常用属性：indices，values，dense\_shape

喂数据(tf.sparse\_placeholder)：

sp = tf.sparse\_placeholder(tf.int64)

sess.run(xxx, feed\_dict={sp: (indices, values, dense\_shape)})

代码示例

import tensorflow as tf

a = tf.SparseTensor(indices=[[0, 0], [1, 2]], values=[1, 2], dense\_shape=[3, 4])

[[1, 0, 0, 0]

[0, 0, 2, 0]

[0, 0, 0, 0]]

with tf.Session() as sess:

b = sess.run(a)

print(b)

print(b.indices)

print(b.values)

print(b.dense\_shape)

>>>SparseTensorValue(indices=array([[0, 0], [1, 2]], dtype=int64), values=array([1, 2]), dense\_shape=array([3, 4], dtype=int64))

>>>[[0 0]

[1 2]]

>>>[1 2]

>>>[3 4]

**2.** **生成 SparseTensor（sparse\_tuple\_from）：**

TensorFlow 中没有现成的函数，可以自己封装起来，以备不时之需。

import numpy as np

import tensorflow as tf

#转化一个序列列表为稀疏矩阵，这个用的多

def sparse\_tuple\_from(sequences, dtype=np.int32):

    """

    Create a sparse representention of x.

    Args:

        sequences: a list of lists of type dtype where each element is a sequence

    Returns:

        A tuple with (indices, values, shape)

    """

    indices = []

    values = []

    for n, seq in enumerate(sequences):

        # zip打包为元组

        indices.extend(zip([n] \* len(seq), range(len(seq))))

        values.extend(seq)

    # indices:二维int64的矩阵，代表非0的坐标点

    indices = np.asarray(indices, dtype=np.int64)

    # values:二维tensor，代表indice位置的数据值

    values = np.asarray(values, dtype=dtype)

    #dense\_shape:一维，代表稀疏矩阵的大小

    shape = np.asarray([len(sequences), np.asarray(indices).max(0)[1] + 1], dtype=np.int64)

    return indices, values, shape

**3.** SparseTensor 转 DenseTensor（tensorflow自带）：

Def tf.sparse\_tensor\_to\_dense(sp\_input, default\_value=0, validate\_indices=True, name=None):

输入参数：

sp\_input： 输入的 SparseTensor

default\_value： Scalar value to set for indices not specified in sp\_input，默认为 0

validate\_indices： A boolean value. If True, indices are checked to make sure they are sorted in lexicographic order and that there are no repeats in indices

输出：

A dense tensor with shape sp\_input.dense\_shape and values specified by the non-empty values in sp\_input， Indices not in sp\_input are assigned default\_value

**代码示例：**

import tensorflow as tf

a = tf.SparseTensor(indices=[[0, 1], [0, 3], [2, 0]], values=[2, 3, 5], dense\_shape=[3, 5])

# 未指定索引的位置使用 -1 来填充

b = tf.sparse\_tensor\_to\_dense(a, default\_value=-1)

with tf.Session() as sess:

print(sess.run(b))

[[-1 2 -1 3 -1]

[-1 -1 -1 -1 -1]

[ 5 -1 -1 -1 -1]]

**4.** SparseTensor 转 DenseTensor（自定义）：

#转化一个稀疏矩阵为序列列表

def decode\_sparse\_tensor(sparse\_tensor):

    #print("sparse\_tensor = ", sparse\_tensor)

    decoded\_indexes = list()

    current\_i = 0

    current\_seq = []

    for offset, i\_and\_index in enumerate(sparse\_tensor[0]):

        i = i\_and\_index[0]

        if i != current\_i:

            decoded\_indexes.append(current\_seq)

            current\_i = i

            current\_seq = list()

        current\_seq.append(offset)

    decoded\_indexes.append(current\_seq)

    #print("decoded\_indexes = ", decoded\_indexes)

    #准备输出结果

    result = []

    for index in decoded\_indexes:

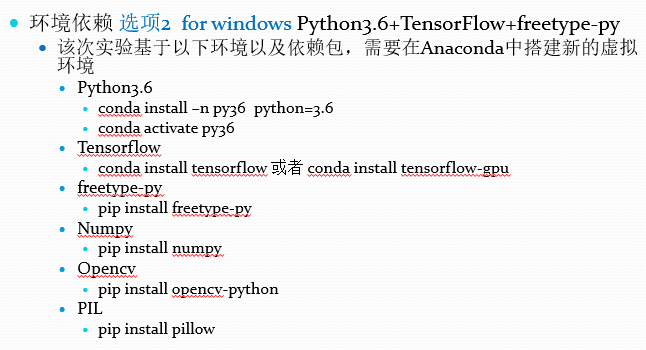
        #print("index = ", index)

        result.append(decode\_a\_seq(index, sparse\_tensor))

        #print(result)

    return result

1. **实验环境**



1. **实验程序**

**参数定义**

#定义一些常量

#图片大小，32 x 256

OUTPUT\_SHAPE = (32,256)

#训练最大轮次

num\_epochs = 10000

#LSTM的cell的h输出为64\*1的向量

num\_hidden = 64

#一层LSTM

num\_layers = 1

#获取了身份证号对象

obj = gen\_id\_card()

#识别对象集合

num\_classes = obj.len + 1 + 1  # 10位数字 + blank + ctc blank

#初始化学习速率

INITIAL\_LEARNING\_RATE = 1e-3

#每5000步衰减一次学习率

DECAY\_STEPS = 5000

#100步汇报一次

REPORT\_STEPS = 100

#学习率衰减系数0.9

LEARNING\_RATE\_DECAY\_FACTOR = 0.9  # The learning rate decay factor

MOMENTUM = 0.9

#数字字符串

DIGITS='0123456789'

BATCHES = 10

BATCH\_SIZE = 128

TRAIN\_SIZE = BATCHES \* BATCH\_SIZE

**获取身份证号码图片：**

class gen\_id\_card(object):

    def \_\_init\_\_(self):

       #self.words = open('AllWords.txt', 'r').read().split(' ')

       self.number = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

       self.char\_set = self.number

       #self.char\_set = self.words + self.number

       self.len = len(self.char\_set)

       self.max\_size = 18

       self.ft = put\_chinese\_text('fonts/OCR-B.ttf')

    #随机生成字串，长度固定

    #返回text,及对应的向量

    def random\_text(self):

        text = ''

        vecs = np.zeros((self.max\_size \* self.len))

        size = random.randint(1, self.max\_size)

        #size = self.max\_size

        for i in range(size):

            c = random.choice(self.char\_set)

            vec = self.char2vec(c)

            text = text + c

            vecs[i\*self.len:(i+1)\*self.len] = np.copy(vec)

        return text,vecs

    #根据生成的text，生成image,返回标签和图片元素数据

    def gen\_image(self):

        text,vec = self.random\_text()

        img = np.zeros([32,256,3])

        color\_ = (255,255,255) # Write

        pos = (0, 0)

        text\_size = 21

        image = self.ft.draw\_text(img, pos, text, text\_size, color\_)

        return image[:,:,2],text,vec

    #单字转向量

    def char2vec(self, c):

        vec = np.zeros((self.len))

        for j in range(self.len):

            if self.char\_set[j] == c:

                vec[j] = 1

        return vec

    #向量转文本

    def vec2text(self, vecs):

        text = ''

        v\_len = len(vecs)

        for i in range(v\_len):

            if(vecs[i] == 1):

                text = text + self.char\_set[i % self.len]

        return text

生成的不定长数字字符串：





**获取字母图片：**

if \_\_name\_\_ == '\_\_main\_\_':

    '''

    genObj = gen\_id\_card()

    image\_data,label,vec = genObj.gen\_image()

    cv2.imshow('image', image\_data)

    cv2.waitKey(0)

    '''

    line = 'RNN-LSTM+CTC'

    img = np.zeros([32,256,3])

    color\_ = (255,255,255) # Green

    pos = (0, 0)

    text\_size = 20

    no = put\_chinese\_text('fonts/OCR-B.ttf')

    image = no.draw\_text(img, pos, line, text\_size, color\_)

    cv2.imshow('image', image)

    cv2.waitKey(0)

    def draw\_text(self, image, pos, text, text\_size, text\_color):

        '''

        draw chinese(or not) text with ttf

        :param image:     image(numpy.ndarray) to draw text

        :param pos:       where to draw text

        :param text:      the context, for chinese should be unicode type

        :param text\_size: text size

        :param text\_color:text color

        :return:          image

        '''

        self.\_face.set\_char\_size(text\_size \* 64)

        metrics = self.\_face.size

        ascender = metrics.ascender/64.0

        #descender = metrics.descender/64.0

        #height = metrics.height/64.0

        #linegap = height - ascender + descender

        ypos = int(ascender)

        #text = text.decode('utf-8')

        img = self.draw\_string(image, pos[0], pos[1]+ypos, text, text\_color)

        return img

**LSTM模型定义**

def get\_train\_model():

    #features = convolutional\_layers()

    #print features.get\_shape()

    # (?,?,32)

    inputs = tf.placeholder(tf.float32, [None, None, OUTPUT\_SHAPE[0]])

    #定义ctc\_loss需要的稀疏矩阵

    targets = tf.sparse\_placeholder(tf.int32)

    #1维向量 序列长度 [batch\_size,]

    seq\_len = tf.placeholder(tf.int32, [None])

    #定义LSTM网络

    cell = tf.contrib.rnn.LSTMCell(num\_hidden, state\_is\_tuple=True)

    stack = tf.contrib.rnn.MultiRNNCell([cell] \* num\_layers, state\_is\_tuple=True)

    outputs, \_ = tf.nn.dynamic\_rnn(cell, inputs, seq\_len, dtype=tf.float32)

    shape = tf.shape(inputs)

    batch\_s, max\_timesteps = shape[0], shape[1]#128，256

    outputs = tf.reshape(outputs, [-1, num\_hidden])#128

    W = tf.Variable(tf.truncated\_normal([num\_hidden,#128 12

                                          num\_classes],

                                         stddev=0.1), name="W")

    b = tf.Variable(tf.constant(0., shape=[num\_classes]), name="b")#12

    # 输出×权重+偏置

    logits = tf.matmul(outputs, W) + b

    # 128 256 12

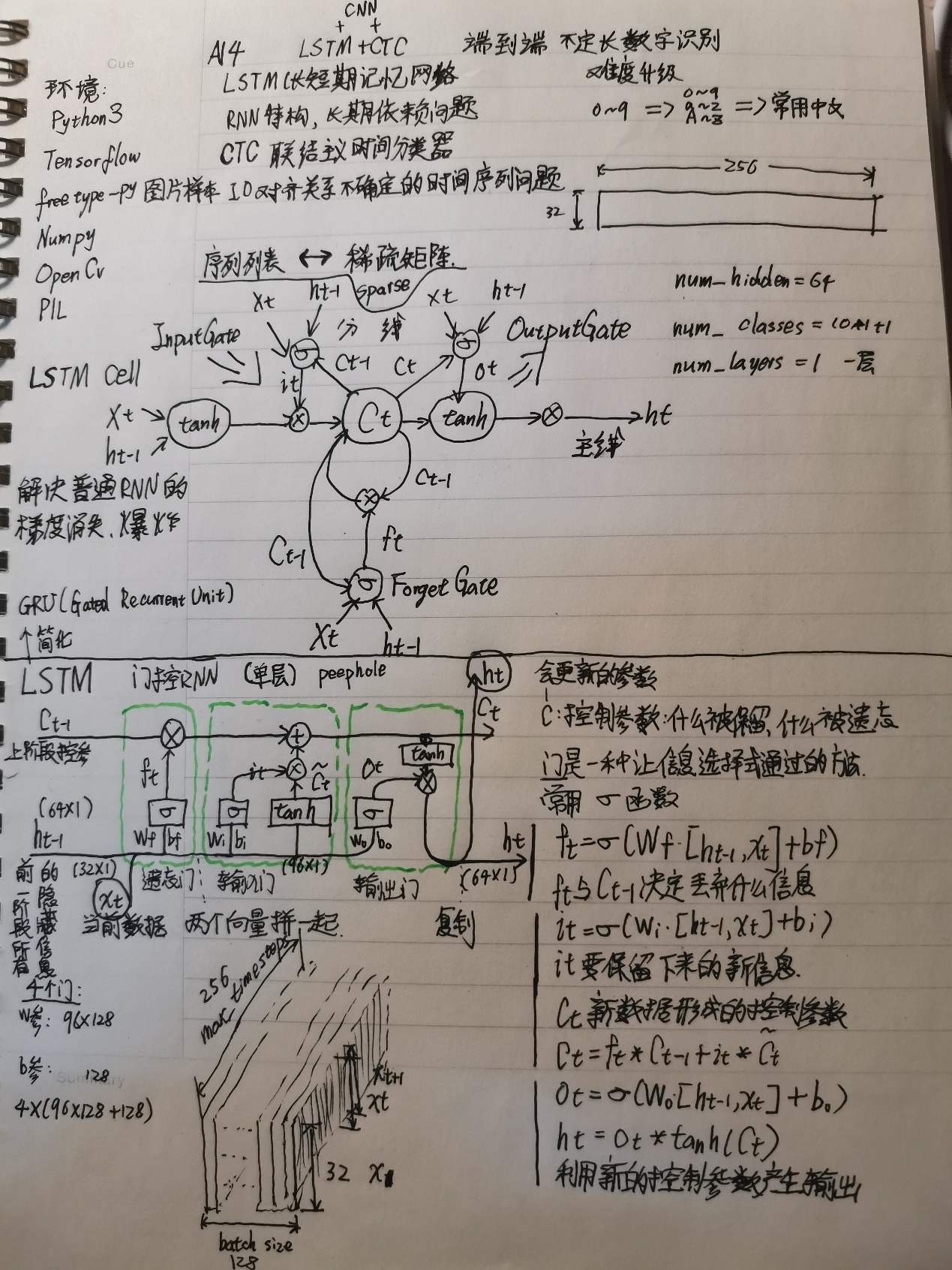
    logits = tf.reshape(logits, [batch\_s, -1, num\_classes])

    # 256 128 12

    logits = tf.transpose(logits, (1, 0, 2))

    #

return logits, inputs, targets, seq\_len, W, b

**CTC Loss**

    loss = tf.nn.ctc\_loss(labels=targets,inputs=logits, sequence\_length=seq\_len)

    cost = tf.reduce\_mean(loss)

    #optimizer = tf.train.MomentumOptimizer(learning\_rate=learning\_rate,momentum=MOMENTUM).minimize(cost, global\_step=global\_step)

#优化方法Adam

optimizer = tf.train.AdamOptimizer(learning\_rate=learning\_rate).minimize(loss,global\_step=global\_step)

#解码方法beam\_search

    decoded, log\_prob = tf.nn.ctc\_beam\_search\_decoder(logits, seq\_len, merge\_repeated=False)

**训练**

    def do\_batch():

        train\_inputs, train\_targets, train\_seq\_len = get\_next\_batch(BATCH\_SIZE)

        feed = {inputs: train\_inputs, targets: train\_targets, seq\_len: train\_seq\_len}

        b\_loss,b\_targets, b\_logits, b\_seq\_len,b\_cost, steps, \_ = session.run([loss, targets, logits, seq\_len, cost, global\_step, optimizer], feed)

        #print b\_loss

        #print b\_targets, b\_logits, b\_seq\_len

        print(b\_cost, steps)

        if steps > 0 and steps % REPORT\_STEPS == 0:

#当准确率大于0.9，停止训练

            if(do\_report()>0.9):

                save\_path = saver.save(session, "./ocr.model", global\_step=steps)

                print(save\_path)

                raise FError("Train succcess")

        return b\_cost, steps

**报告准确率**

    def do\_report():

        test\_inputs,test\_targets,test\_seq\_len = get\_next\_batch(BATCH\_SIZE)

        test\_feed = {inputs: test\_inputs,

                     targets: test\_targets,

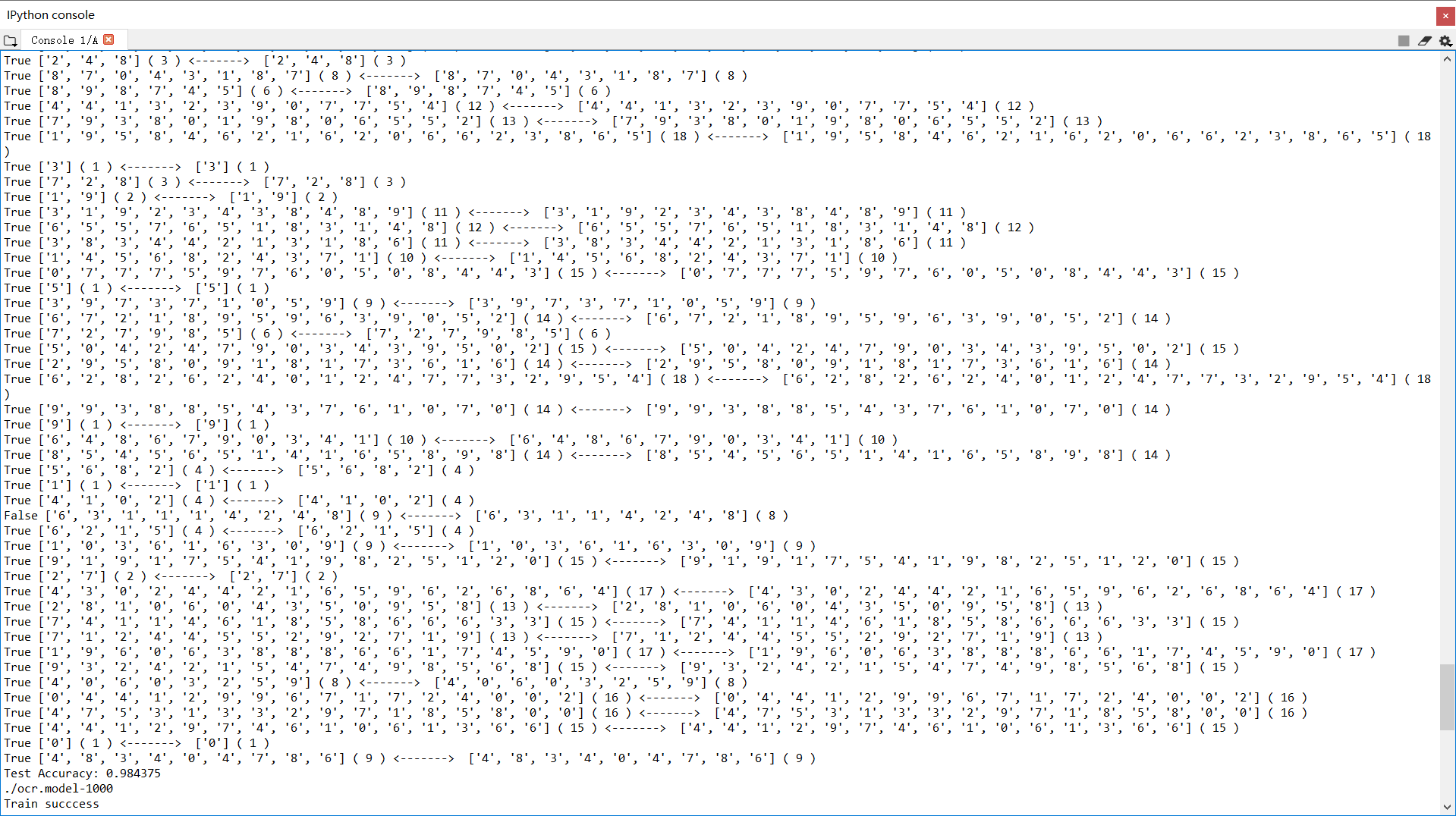
                     seq\_len: test\_seq\_len}

        dd, log\_probs, accuracy = session.run([decoded[0], log\_prob, acc], test\_feed)

        return report\_accuracy(dd, test\_targets)

        # decoded\_list = decode\_sparse\_tensor(dd)

**训练完成：**

****

**预测（beam\_search）：**

def crack\_image():

    global\_step = tf.Variable(0, trainable=False)

    learning\_rate = tf.train.exponential\_decay(INITIAL\_LEARNING\_RATE,

                                                global\_step,

                                                DECAY\_STEPS,

                                                LEARNING\_RATE\_DECAY\_FACTOR,

                                                staircase=True)

    logits, inputs, targets, seq\_len, W, b = get\_train\_model()

    decoded, log\_prob = tf.nn.ctc\_beam\_search\_decoder(logits, seq\_len, merge\_repeated=False)

    acc = tf.reduce\_mean(tf.edit\_distance(tf.cast(decoded[0], tf.int32), targets))

    init = tf.global\_variables\_initializer()

    saver = tf.train.Saver()

with tf.Session() as session:

#加载模型

       saver.restore(session, "./ocr.model-1200")

       #test\_inputs,test\_targets,test\_seq\_len = get\_next\_batch(1)

       test\_inputs,test\_targets,test\_seq\_len,image = get\_a\_image()

       test\_feed = {inputs: test\_inputs,

                    targets: test\_targets,

                    seq\_len: test\_seq\_len}

       dd, log\_probs, accuracy = session.run([decoded[0], log\_prob, acc], test\_feed)

       report\_accuracy(dd, test\_targets)

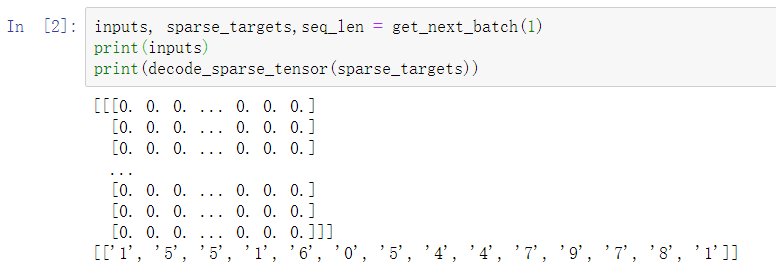
       cv2.imshow('image', image)

       cv2.waitKey(0)

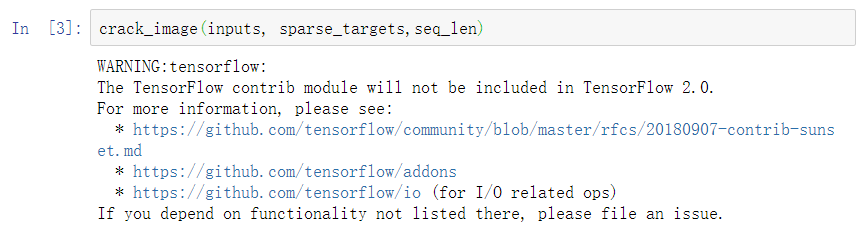
为了验证一张图片的破解情况，我改写了一下crack\_image()函数，使之带有参数

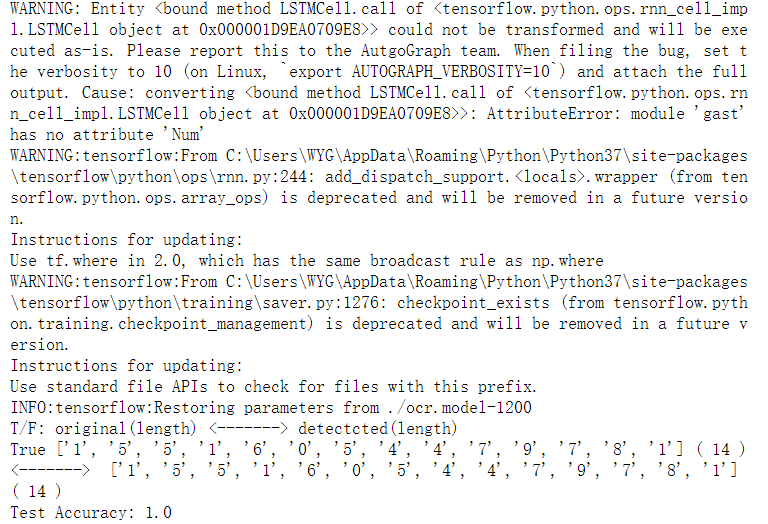


查看inputs和解码后的输出

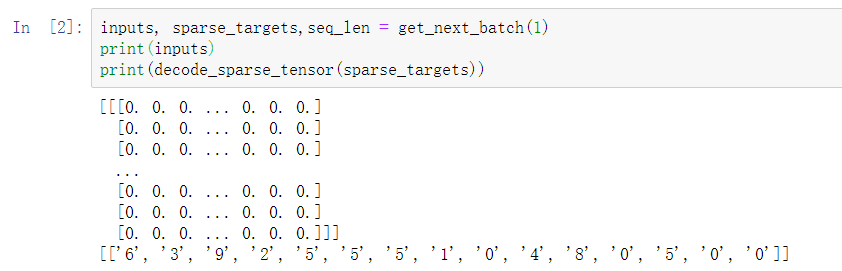


破解图片

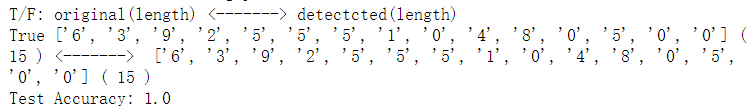




预测正确







模型已经具有识别不定长数字串的能力，实验结束。