https://zhuanlan.zhihu.com/p/26078299



**使用深度学习来破解 captcha 验证码**

[杨培文](https://www.zhihu.com/people/yangpw)[杨培文](https://www.zhihu.com/people/yangpw)

12 天前

本文会通过 [Keras](http://link.zhihu.com/?target=https%3A//keras.io/" \t "_blank) 搭建一个深度卷积神经网络来识别 captcha 验证码，建议使用显卡来运行该项目。

下面的可视化代码都是在 jupyter notebook 中完成的，如果你希望写成 python 脚本，稍加修改即可正常运行，当然也可以去掉这些可视化代码。Keras 版本：1.2.2。

GitHub 地址：[https://github.com/ypwhs/captcha\_break](http://link.zhihu.com/?target=https%3A//github.com/ypwhs/captcha_break" \t "_blank)

**captcha**

captcha 是用 python 写的生成验证码的库，它支持图片验证码和语音验证码，我们使用的是它生成图片验证码的功能。

首先我们设置我们的验证码格式为数字加大写字母，生成一串验证码试试看：

**from** captcha.image **import** ImageCaptcha

**import** matplotlib.pyplot **as** plt

**import** numpy **as** np

**import** random

**%**matplotlib inline

**%**config InlineBackend**.**figure\_format **=** 'retina'

**import** string

characters **=** string**.**digits **+** string**.**ascii\_uppercase

**print**(characters)

width, height, n\_len, n\_class **=** 170, 80, 4, len(characters)

generator **=** ImageCaptcha(width**=**width, height**=**height)

random\_str **=** ''**.**join([random**.**choice(characters) **for** j **in** range(4)])

img **=** generator**.**generate\_image(random\_str)

plt**.**imshow(img)

plt**.**title(random\_str)



**数据生成器**

训练模型的时候，我们可以选择两种方式来生成我们的训练数据，一种是一次性生成几万张图，然后开始训练，一种是定义一个数据生成器，然后利用 fit\_generator 函数来训练。

第一种方式的好处是训练的时候显卡利用率高，如果你需要经常调参，可以一次生成，多次使用；第二种方式的好处是你不需要生成大量数据，训练过程中可以利用 CPU 生成数据，而且还有一个好处是你可以无限生成数据。

我们的数据格式如下：

**X**

X 的形状是 (batch\_size, height, width, 3)，比如一批生成32个样本，图片宽度为170，高度为80，那么形状就是 (32, 80, 170, 3)，取第一张图就是 X[0]。

**y**

y 的形状是四个 (batch\_size, n\_class)，如果转换成 numpy 的格式，则是 (n\_len, batch\_size, n\_class)，比如一批生成32个样本，验证码的字符有36种，长度是4位，那么它的形状就是4个 (32, 36)，也可以说是 (4, 32, 36)，解码函数在下个代码块。

**def** **gen**(batch\_size**=**32):

X **=** np**.**zeros((batch\_size, height, width, 3), dtype**=**np**.**uint8)

y **=** [np**.**zeros((batch\_size, n\_class), dtype**=**np**.**uint8) **for** i **in** range(n\_len)]

generator **=** ImageCaptcha(width**=**width, height**=**height)

**while** True:

**for** i **in** range(batch\_size):

random\_str **=** ''**.**join([random**.**choice(characters) **for** j **in** range(4)])

X[i] **=** generator**.**generate\_image(random\_str)

**for** j, ch **in** enumerate(random\_str):

y[j][i, :] **=** 0

y[j][i, characters**.**find(ch)] **=** 1

**yield** X, y

上面就是一个可以无限生成数据的例子，我们将使用这个生成器来训练我们的模型。

**使用生成器**

生成器的使用方法很简单，只需要用 next 函数即可。下面是一个例子，生成32个数据，然后显示第一个数据。当然，在这里我们还对生成的 One-Hot 编码后的数据进行了解码，首先将它转为 numpy 数组，然后取36个字符中最大的数字的位置，因为神经网络会输出36个字符的概率，然后将概率最大的四个字符的编号转换为字符串。

**def** **decode**(y):

y **=** np**.**argmax(np**.**array(y), axis**=**2)[:,0]

**return** ''**.**join([characters[x] **for** x **in** y])

X, y **=** next(gen(1))

plt**.**imshow(X[0])

plt**.**title(decode(y))

**构建深度卷积神经网络**

from keras.models import \*

from keras.layers import \*

input\_tensor = Input((height, width, 3))

x = input\_tensor

for i in range(4):

x = Convolution2D(32\*2\*\*i, 3, 3, activation='relu')(x)

x = Convolution2D(32\*2\*\*i, 3, 3, activation='relu')(x)

x = MaxPooling2D((2, 2))(x)

x = Flatten()(x)

x = Dropout(0.25)(x)

x = [Dense(n\_class, activation='softmax', name='c%d'%(i+1))(x) for i in range(4)]

model = Model(input=input\_tensor, output=x)

model.compile(loss='categorical\_crossentropy',

optimizer='adadelta',

metrics=['accuracy'])

模型结构很简单，特征提取部分使用的是两个卷积，一个池化的结构，这个结构是学的 VGG16 的结构。之后我们将它 Flatten，然后添加 Dropout ，尽量避免过拟合问题，最后连接四个分类器，每个分类器是36个神经元，输出36个字符的概率。

**模型可视化**

得益于 Keras 自带的可视化，我们可以使用几句代码来可视化模型的结构：

**from** keras.utils.visualize\_util **import** plot

**from** IPython.display **import** Image

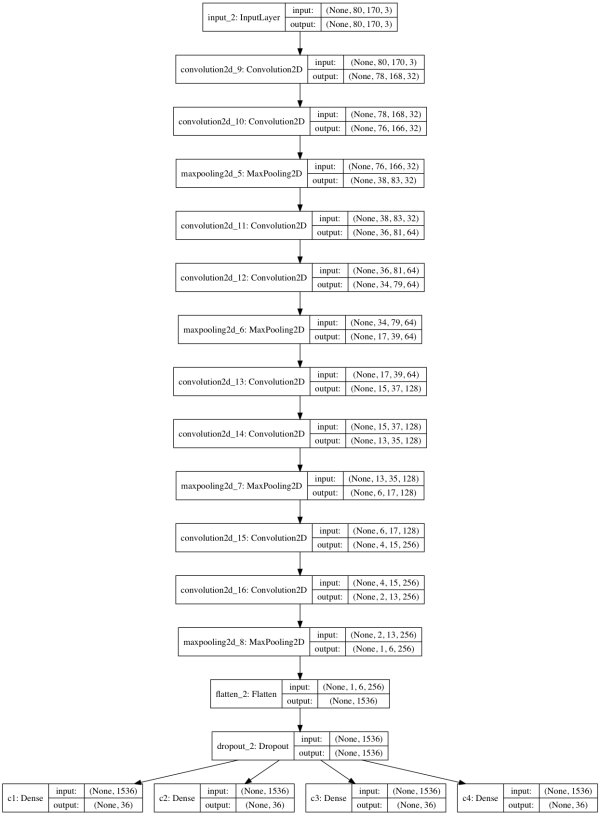
plot(model, to\_file**=**"model.png", show\_shapes**=**True)

Image('model.png')

这里需要使用 pydot 这个库，以及 graphviz 这个库，在 macOS 系统上安装方法如下：

brew install graphviz

pip install pydot-ng



我们可以看到最后一层卷积层输出的形状是 (1, 6, 256)，已经不能再加卷积层了。

**训练模型**

训练模型反而是所有步骤里面最简单的一个，直接使用 model.fit\_generator 即可，这里的验证集使用了同样的生成器，由于数据是通过生成器随机生成的，所以我们不用考虑数据是否会重复。注意，这段代码在笔记本上可能要耗费一下午时间。如果你想让模型预测得更准确，可以将 nb\_epoch改为 10 或者 20，但它也将耗费成倍的时间。注意我们这里使用了一个小技巧，添加 nb\_worker=2 参数让 Keras 自动实现多进程生成数据，摆脱 python 单线程效率低的缺点。如果不添加，耗时120秒，添加则只需80秒。

model**.**fit\_generator(gen(), samples\_per\_epoch**=**51200, nb\_epoch**=**5,

nb\_worker**=**2, pickle\_safe**=**True,

validation\_data**=**gen(), nb\_val\_samples**=**1280)

**测试模型**

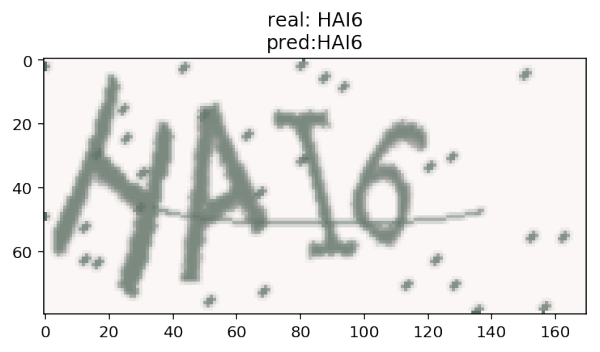
当我们训练完成以后，可以识别一个验证码试试看：

X, y **=** next(gen(1))

y\_pred **=** model**.**predict(X)

plt**.**title('real: %s\npred:%s'**%**(decode(y), decode(y\_pred)))

plt**.**imshow(X[0], cmap**=**'gray')



**计算模型总体准确率**

模型在训练的时候只会显示第几个字符的准确率，为了统计模型的总体准确率，我们可以写下面的函数：

**from** tqdm **import** tqdm

**def** **evaluate**(model, batch\_num**=**20):

batch\_acc **=** 0

generator **=** gen()

**for** i **in** tqdm(range(batch\_num)):

X, y **=** next(generator)

y\_pred **=** model**.**predict(X)

y\_pred **=** np**.**argmax(y\_pred, axis**=**2)**.**T

y\_true **=** np**.**argmax(y, axis**=**2)**.**T

batch\_acc **+=** np**.**mean(map(np**.**array\_equal, y\_true, y\_pred))

**return** batch\_acc **/** batch\_num

evaluate(model)

这里用到了一个库叫做 tqdm，它是一个进度条的库，为的是能够实时反馈进度。然后我们通过一些 numpy 计算去统计我们的准确率，这里计算规则是只要有一个错，那么就不算它对。经过计算，我们的模型的总体准确率在经过五代训练就可以达到 90%，继续训练还可以达到更高的准确率。

**模型总结**

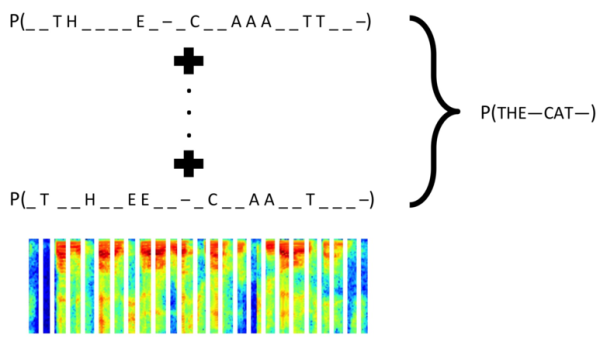
模型的大小是16MB，在我的笔记本上跑1000张验证码需要用20秒，当然，显卡会更快。对于验证码识别的问题来说，哪怕是10%的准确率也已经称得上破解，毕竟假设100%识别率破解要一个小时，那么10%的识别率也只用十个小时，还算等得起，而我们的识别率有90%，已经可以称得上完全破解了这类验证码。

**改进**

对于这种按顺序书写的文字，我们还有一种方法可以使用，那就是循环神经网络来识别序列。下面我们来了解一下如何使用循环神经网络来识别这类验证码。

**CTC Loss**

这个 loss 是一个特别神奇的 loss，它可以在只知道序列的顺序，不知道具体位置的情况下，让模型收敛。在这方面百度似乎做得很不错，利用它来识别音频信号。（[warp-ctc](http://link.zhihu.com/?target=https%3A//github.com/baidu-research/warp-ctc)）



那么在 Keras 里面，CTC Loss 已经内置了，我们直接定义这样一个函数，即可实现 CTC Loss，由于我们使用的是循环神经网络，所以默认丢掉前面两个输出，因为它们通常无意义，且会影响模型的输出。

* y\_pred 是模型的输出，是按顺序输出的37个字符的概率，因为我们这里用到了循环神经网络，所以需要一个空白字符的概念；
* labels 是验证码，是四个数字；
* input\_length 表示 y\_pred 的长度，我们这里是15；
* label\_length 表示 labels 的长度，我们这里是4。

**from** keras **import** backend **as** K

**def** **ctc\_lambda\_func**(args):

y\_pred, labels, input\_length, label\_length **=** args

y\_pred **=** y\_pred[:, 2:, :]

**return** K**.**ctc\_batch\_cost(labels, y\_pred, input\_length, label\_length)

**模型结构**

我们的模型结构是这样设计的，首先通过卷积神经网络去识别特征，然后经过一个全连接降维，再按水平顺序输入到一种特殊的循环神经网络，叫 GRU，它具有一些特殊的性质，为什么用 GRU 而不用 LSTM 呢？总的来说就是它的效果比 LSTM 好，所以我们用它。

**from** keras.models **import** **\***

**from** keras.layers **import** **\***

rnn\_size **=** 128

input\_tensor **=** Input((width, height, 3))

x **=** input\_tensor

**for** i **in** range(3):

x **=** Convolution2D(32, 3, 3, activation**=**'relu')(x)

x **=** Convolution2D(32, 3, 3, activation**=**'relu')(x)

x **=** MaxPooling2D(pool\_size**=**(2, 2))(x)

conv\_shape **=** x**.**get\_shape()

x **=** Reshape(target\_shape**=**(int(conv\_shape[1]), int(conv\_shape[2]**\***conv\_shape[3])))(x)

x **=** Dense(32, activation**=**'relu')(x)

gru\_1 **=** GRU(rnn\_size, return\_sequences**=**True, init**=**'he\_normal', name**=**'gru1')(x)

gru\_1b **=** GRU(rnn\_size, return\_sequences**=**True, go\_backwards**=**True,

init**=**'he\_normal', name**=**'gru1\_b')(x)

gru1\_merged **=** merge([gru\_1, gru\_1b], mode**=**'sum')

gru\_2 **=** GRU(rnn\_size, return\_sequences**=**True, init**=**'he\_normal', name**=**'gru2')(gru1\_merged)

gru\_2b **=** GRU(rnn\_size, return\_sequences**=**True, go\_backwards**=**True,

init**=**'he\_normal', name**=**'gru2\_b')(gru1\_merged)

x **=** merge([gru\_2, gru\_2b], mode**=**'concat')

x **=** Dropout(0.25)(x)

x **=** Dense(n\_class, init**=**'he\_normal', activation**=**'softmax')(x)

base\_model **=** Model(input**=**input\_tensor, output**=**x)

labels **=** Input(name**=**'the\_labels', shape**=**[n\_len], dtype**=**'float32')

input\_length **=** Input(name**=**'input\_length', shape**=**[1], dtype**=**'int64')

label\_length **=** Input(name**=**'label\_length', shape**=**[1], dtype**=**'int64')

loss\_out **=** Lambda(ctc\_lambda\_func, output\_shape**=**(1,),

name**=**'ctc')([x, labels, input\_length, label\_length])

model **=** Model(input**=**[input\_tensor, labels, input\_length, label\_length], output**=**[loss\_out])

model**.**compile(loss**=**{'ctc': **lambda** y\_true, y\_pred: y\_pred}, optimizer**=**'adadelta')

**模型可视化**

可视化的代码同上，这里只贴图。

可以看到模型比上一个模型复杂了许多，但实际上只是因为输入比较多，所以它显得很大。还有一个值得注意的地方，我们的图片在输入的时候是经过了旋转的，这是因为我们希望以水平方向输入，而图片在 numpy 里默认是这样的形状：(height, width, 3)，因此我们使用了 transpose 函数将图片转为了(width, height, 3)的格式，然后经过各种卷积和降维，变成了 (17, 32)，这里的每个长度为32的向量都代表一个竖条的图片的特征，从左到右，一共有17条。然后我们兵分两路，一路从左到右输入到 GRU，一路从右到左输入到 GRU，然后将他们输出的结果加起来。再兵分两路，还是一路正方向，一路反方向，只不过第二次我们直接将它们的输出连起来，然后经过一个全连接，输出每个字符的概率。

**数据生成器**

**def** **gen**(batch\_size**=**128):

X **=** np**.**zeros((batch\_size, width, height, 3), dtype**=**np**.**uint8)

y **=** np**.**zeros((batch\_size, n\_len), dtype**=**np**.**uint8)

**while** True:

generator **=** ImageCaptcha(width**=**width, height**=**height)

**for** i **in** range(batch\_size):

random\_str **=** ''**.**join([random**.**choice(characters) **for** j **in** range(4)])

X[i] **=** np**.**array(generator**.**generate\_image(random\_str))**.**transpose(1, 0, 2)

y[i] **=** [characters**.**find(x) **for** x **in** random\_str]

**yield** [X, y, np**.**ones(batch\_size)**\***int(conv\_shape[1]**-**2),

np**.**ones(batch\_size)**\***n\_len], np**.**ones(batch\_size)

**评估模型**

**def** **evaluate**(model, batch\_num**=**10):

batch\_acc **=** 0

generator **=** gen()

**for** i **in** range(batch\_num):

[X\_test, y\_test, \_, \_], \_ **=** next(generator)

y\_pred **=** base\_model**.**predict(X\_test)

shape **=** y\_pred[:,2:,:]**.**shape

ctc\_decode **=** K**.**ctc\_decode(y\_pred[:,2:,:],

input\_length**=**np**.**ones(shape[0])**\***shape[1])[0][0]

out **=** K**.**get\_value(ctc\_decode)[:, :4]

**if** out**.**shape[1] **==** 4:

batch\_acc **+=** ((y\_test **==** out)**.**sum(axis**=**1) **==** 4)**.**mean()

**return** batch\_acc **/** batch\_num

我们会通过这个函数来评估我们的模型，和上面的评估标准一样，只有全部正确，我们才算预测正确，中间有个坑，就是模型最开始训练的时候，并不一定会输出四个字符，所以我们如果遇到所有的字符都不到四个的时候，就不计算了，相当于加0，遇到多于4个字符的时候，只取前四个。

**评估回调**

因为 Keras 没有针对这种输出计算准确率的选项，因此我们需要自定义一个回调函数，它会在每一代训练完成的时候计算模型的准确率。

**from** keras.callbacks **import** **\***

**class** **Evaluate**(Callback):

**def** **\_\_init\_\_**(self):

self**.**accs **=** []

**def** **on\_epoch\_end**(self, epoch, logs**=**None):

acc **=** evaluate(base\_model)**\***100

self**.**accs**.**append(acc)

**print**

**print** 'acc: %f%%'**%**acc

evaluator **=** Evaluate()

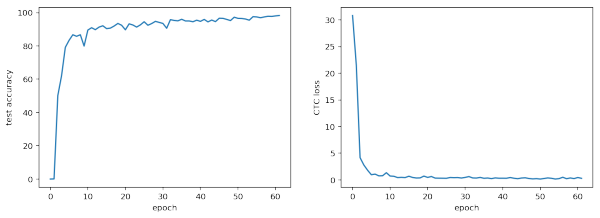
**训练模型**

由于 CTC Loss 收敛很慢，所以我们需要设置比较大的代数，这里我们设置了100代，然后添加了一个早期停止的回调和我们上面定义的回调，但是第一次训练只训练37代就停了，测试准确率才95%，我又在这个基础上继续训练了一次，停在了25代，得到了98%的准确率，所以一共训练了62代。

model.fit\_generator(gen(), samples\_per\_epoch=51200, nb\_epoch=100,

callbacks=[evaluator],

nb\_worker=2, pickle\_safe=True)



**测试模型**

characters2 **=** characters **+** ' '

[X\_test, y\_test, \_, \_], \_ **=** next(gen(1))

y\_pred **=** base\_model**.**predict(X\_test)

y\_pred **=** y\_pred[:,2:,:]

out **=** K**.**get\_value(K**.**ctc\_decode(y\_pred, input\_length**=**np**.**ones(y\_pred**.**shape[0])**\***y\_pred**.**shape[1], )[0][0])[:, :4]

out **=** ''**.**join([characters[x] **for** x **in** out[0]])

y\_true **=** ''**.**join([characters[x] **for** x **in** y\_test[0]])

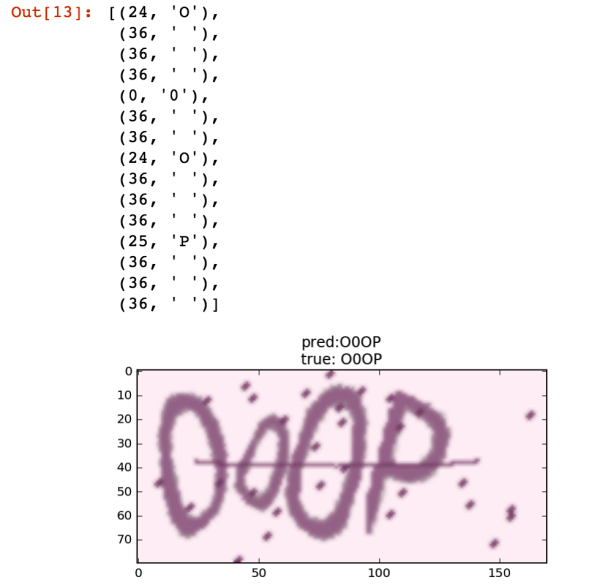
plt**.**imshow(X\_test[0]**.**transpose(1, 0, 2))

plt**.**title('pred:' **+** str(out) **+** '\ntrue: ' **+** str(y\_true))

argmax **=** np**.**argmax(y\_pred, axis**=**2)[0]

list(zip(argmax, ''**.**join([characters2[x] **for** x **in** argmax])))

这次随机出来的验证码很厉害，是O0OP，不过更厉害的是模型认出来了。



**有趣的问题**

我又用之前的模型做了个测试，对于 O0O0 这样丧心病狂的验证码，模型偶尔也能正确识别，这让我非常惊讶，它是真的能识别 O 与 0 的差别呢，还是猜出来的呢？这很难说。

generator **=** ImageCaptcha(width**=**width, height**=**height)

random\_str **=** 'O0O0'

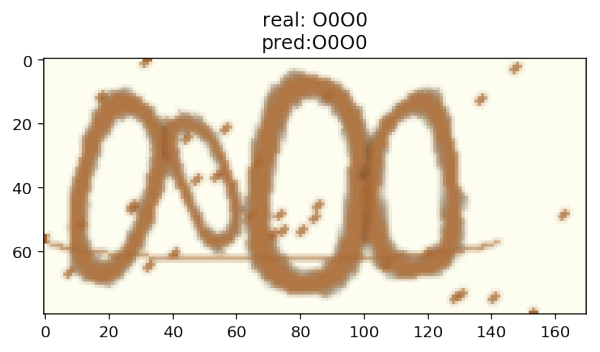
X **=** generator**.**generate\_image(random\_str)

X **=** np**.**expand\_dims(X, 0)

y\_pred **=** model**.**predict(X)

plt**.**title('real: %s\npred:%s'**%**(random\_str, decode(y\_pred)))

plt**.**imshow(X[0], cmap**=**'gray')



**总结**

模型的大小是4.7MB，在我的笔记本上跑1000张验证码需要用14秒，平均一秒识别71张，估计可以拼过网速。至于[深度学习](http://link.zhihu.com/?target=https%3A//cn.udacity.com/course/deep-learning-nanodegree-foundation--nd101)到底能不能识别双胞胎，相信各位已经有了答案。