

基于深度学习的验证码破解、对抗与反对抗研究

概述

在信息安全领域，由于身份鉴别和防止计算机自动攻击的需要，促使了验证码技术的诞生，并伴随着验证码技术对抗的此消彼涨。当前验证码的形式和样式都日益丰富，但其自身的安全性却为很多站点所忽略。本文在利用深度学习破解验证码的基础上，还进行对抗与反对抗研究，并根据实验结果提出验证码改进策略。

验证码诞生

yahoo 邮箱在九几年的时候，业务深受各种邮箱机器人的困扰，存在着大量的垃圾邮件，于是他们找到了当时仍在读大学的路易斯·冯·安(Luis von Ahn)，并设计了经典的图形验证码，即通过简单的扭曲图形文字进行机器的识别。

验证码类型

1. 图片验证码：在网页上以图片形式呈现给用户。
2. 手机验证码：用户在网页上提交自己的手机号码，系统以短信形式将验证码发送到用户手机上。
3. 邮件验证码：用户在网页上提交自己的电子邮箱，系统以电子邮箱形式将验证码发送到用户邮箱上。
4.

验证码沦陷

验证码的本质是要容易被正常用户解答并且让恶意用户无法解答，即将正常用户和恶意用户区分开来。然而各类验证码都已经有了相当成熟的对抗办法，更不用说现在已经泛滥的打码平台了。

1. 图片验证码：
 - A. 预处理
 - a. 背景色混淆 - 二值化
 - b. 字体镂空 - 凸多边形填充
 - c. 多字体混用 - 多字体训练
 - d. 文字扭曲 - 字库
 - e. 文字黏连 - 滴水法分割
 - f. 干扰线 - 联通量阈值去除
 - g.
 - B. 识别
 - a. 字库比较
 - b. 机器学习
 - c. 深度学习
2. 短信验证码：
 - A. 安全性
 - a. 仅从验证码的角度来说，这种方法可以较好地阻挡攻击者。
 - B. 存在问题：
 - a. 受移动运营商短信网关的限制，有时会导致用户无法收到短信。
 - b. 可能造成对手机的 DoS 攻击。
3. 邮件验证码：
 - A. 安全性
 - a. 一般。
 - B. 存在问题
 - a. 攻击者可以编写程序从电子邮箱中获取电子邮件，对邮件分析可以取得验证码。

b. 与手机验证码相似，可能造成对邮箱的 DoS 攻击。

验证码博弈

安全是一个博弈的过程。虽然正义力量和邪恶力量两方数量上略有差距，但是总体战斗力还算是勉强打个平手，互相出招而已。

程序员:熬一晚上升级

攻击者:熬一晚上破解

程序员:熬两晚上升级

攻击者:熬两晚上破解

....(心疼)...

只是随着战火的升级，验证码已经违背了最初设计时的初衷：对普通用户的友好性逐渐消失，普通用户的体验成了这场战争中的牺牲品，这也就造就了一批又一批被用户吐槽的面目全非的验证码。



参考论文

1. 吉治钢 - 基于验证码破解的HTTP攻击原理与防范
2. 文晓阳 高能 夏鲁宁 荆继武 - 高效的验证码识别技术与验证码分类思想

参考链接

1. <http://www.freebuf.com/articles/network/97030.html>
(<http://www.freebuf.com/articles/network/97030.html>)
2. <https://www.cnblogs.com/alisecurity/p/5581049.html>
(<https://www.cnblogs.com/alisecurity/p/5581049.html>)
3. <https://dev.qq.com/topic/581301b146dfb1456904df8d>
(<https://dev.qq.com/topic/581301b146dfb1456904df8d>)

阶段一：验证码破解

参考链接

1. https://github.com/ypwhs/captcha_break (https://github.com/ypwhs/captcha_break)
2. <https://keras.io/> (<https://keras.io/>)
3. <https://opencv.org/> (<https://opencv.org/>)

In []:

```
# 忽略警告  
# import warnings  
# warnings.filterwarnings('ignore')
```

In []:

```
# GPU加速  
# import keras.backend as K  
# cfg = K.tf.ConfigProto()  
# cfg.gpu_options.allow_growth = True  
# K.set_session(K.tf.Session(config = cfg))
```

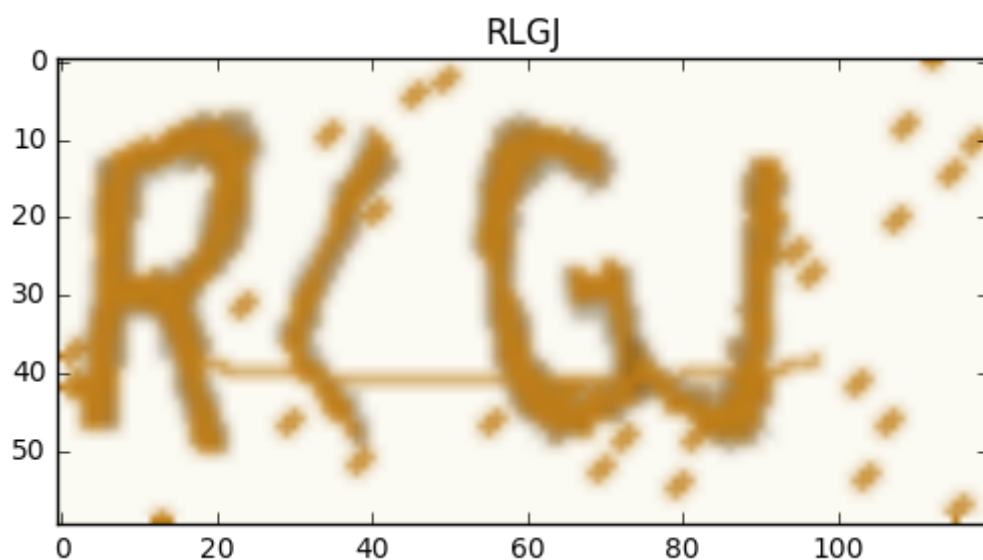
In [1]:

```
%matplotlib inline
from captcha.image import ImageCaptcha # https://pypi.python.org/pypi/captcha/0.
1.1
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
import random, string
import cv2

# 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
characters = string.digits + string.ascii_uppercase
width, height, n_len, n_class = 120, 60, 4, len(characters)
generator = ImageCaptcha(width=width, height=height)
# 测试效果
random_str = ''.join([random.choice(characters) for j in range(4)])
img = generator.generate_image(random_str)
plt.imshow(img)
plt.title(random_str)
```

Out[1]:

<matplotlib.text.Text at 0x7fa47d79fb38>



去噪方案一：形态学转换

腐蚀cv2.erode(img, kernel, iterations = 1)

减少图像中的白色区域（前景）。

1. 用结构元素B，扫描图像A的每一个像素。
2. 用结构元素与其覆盖的二值图像做“与”操作。
3. 如果都为1，结果图像的该像素为1，否则为0。

膨胀cv2.dilate(img, kernel, iterations = 1)

增加图像中的白色区域（前景）。

1. 用结构元素B，扫描图像A的每一个像素。
2. 用结构元素与其覆盖的二值图像做“或”操作。
3. 如果都为0，结果图像的该像素为0，否则为1。

开运算cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)

先腐蚀再膨胀。

闭运算cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)

先膨胀再腐蚀。

In [2]:

```

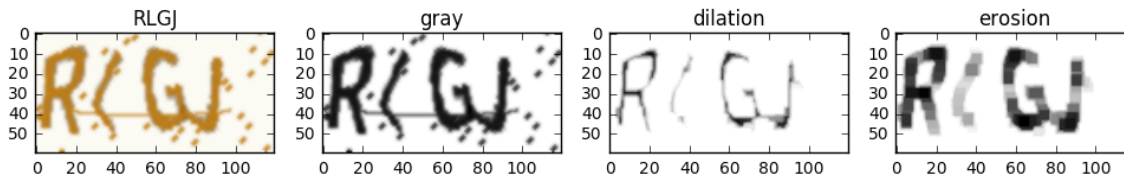
# 灰度化
gray = cv2.cvtColor(np.array(img), cv2.COLOR_BGR2GRAY)
# 定义核
kernel = np.ones((5,5), np.uint8)
# 膨胀
dilation = cv2.dilate(gray, kernel)
# 腐蚀
erosion = cv2.erode(dilation, kernel, iterations = 1)
# 闭运算=膨胀+腐蚀
# closing = cv2.morphologyEx(gray, cv2.MORPH_CLOSE, kernel)

plt.figure(figsize = (12, 8))
plt.subplot(1, 4, 1)
plt.imshow(img, cmap='gray')
plt.title(random_str)
plt.subplot(1, 4, 2)
plt.imshow(gray, cmap='gray')
plt.title('gray')
plt.subplot(1, 4, 3)
plt.imshow(dilation, cmap='gray')
plt.title('dilation')
plt.subplot(1, 4, 4)
plt.imshow(erosion, cmap='gray')
plt.title('erosion')

```

Out[2]:

<matplotlib.text.Text at 0x7fa47c5e5ef0>



去噪方案二：二值化

简单阈值`cv2.threshold(img,127,255,cv2.THRESH_BINARY)`

像素值高于阈值时，我们给这个像素赋予一个新值（可能是白色），否则赋予另外一种颜色。

1. `cv2.THRESH_BINARY`：二值化阈值
2. `cv2.THRESH_BINARY_INV`：反转二值化阈值
3. `cv2.THRESH_TRUNC`：截断二值化阈值
4. `cv2.THRESH_TOZERO`：超过阈值被置为0
5. `cv2.THRESH_TOZERO_INV`：低于阈值被置为0

自适应阈值`cv2.adaptiveThreshold(gray, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)`

根据图像上的每一个小区域计算与其对应的阈值，在同一幅图像上的不同区域采用的是不同的阈值。

1. `cv2.ADPTIVE_THRESH_MEAN_C`：阈值取自相邻区域的平均值
2. `cv2.ADPTIVE_THRESH_GAUSSIAN_C`：阈值取自相邻区域的加权和，权重为一个高斯窗口。

Otsu's 二值化

`cv2.threshold(img,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)`

对一副双峰图像自动根据其直方图计算出一个阈值。

In [3]:

```

# gray = img.convert('L') # 灰度图
# blur = gray.convert('1') # 二值化

# 灰度化
gray = cv2.cvtColor(np.array(img), cv2.COLOR_BGR2GRAY)
# plt.imshow(gray, cmap='gray')

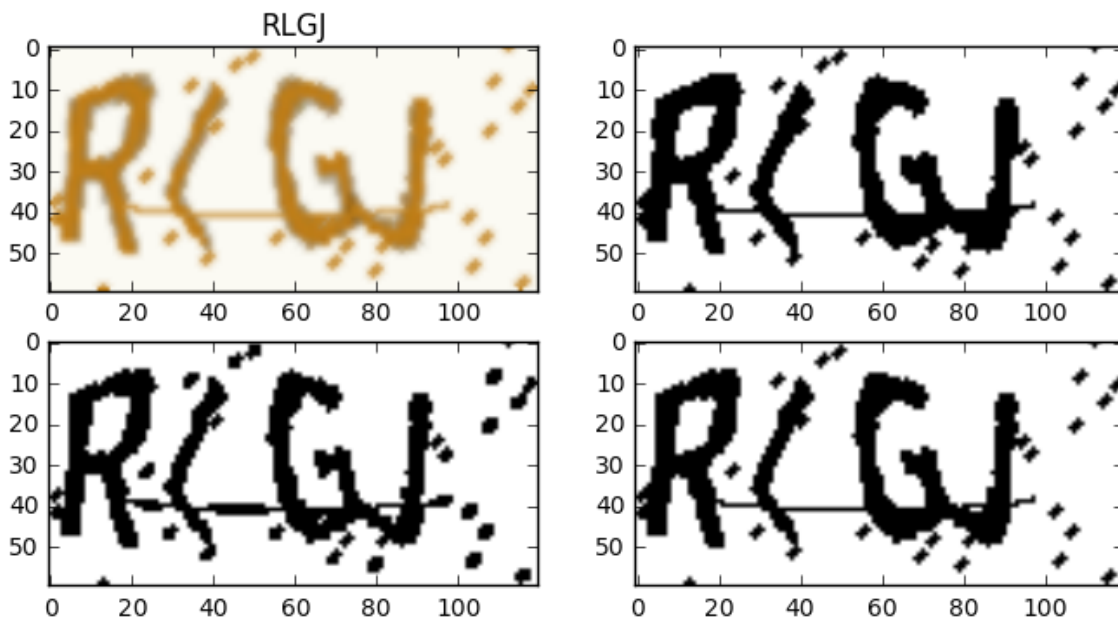
# 二值化
_, blur1 = cv2.threshold(gray, 200, 255, cv2.THRESH_BINARY)
# 自适应二值化
blur2 = cv2.adaptiveThreshold(gray, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)
# Otsu's 二值化
_, blur3 = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY+cv2.THRESH_OTSU)

plt.figure(figsize = (8, 4))
plt.subplot(2, 2, 1)
plt.imshow(img, cmap='gray')
plt.title(random_str)
plt.subplot(2, 2, 2)
plt.imshow(blur1, cmap='gray')
plt.subplot(2, 2, 3)
plt.imshow(blur2, cmap='gray')
plt.subplot(2, 2, 4)
plt.imshow(blur3, cmap='gray')

```

Out[3]:

<matplotlib.image.AxesImage at 0x7fa47c4e5a90>



去噪方案二（接上）：滤波

中值滤波`cv2.medianBlur(img,5)`

用卷积核覆盖区域所有像素的中位数来代替中心像素的值。

均值滤波`cv2.blur(img,(5,5))`

用卷积核覆盖区域所有像素的平均数来代替中心像素的值。

高斯滤波`cv2.GaussianBlur(img,(5,5),0)`

把卷积核换成高斯核（卷积核的值符合高斯分布），原来的求平均数变成求加权平均数

In [4]:

```

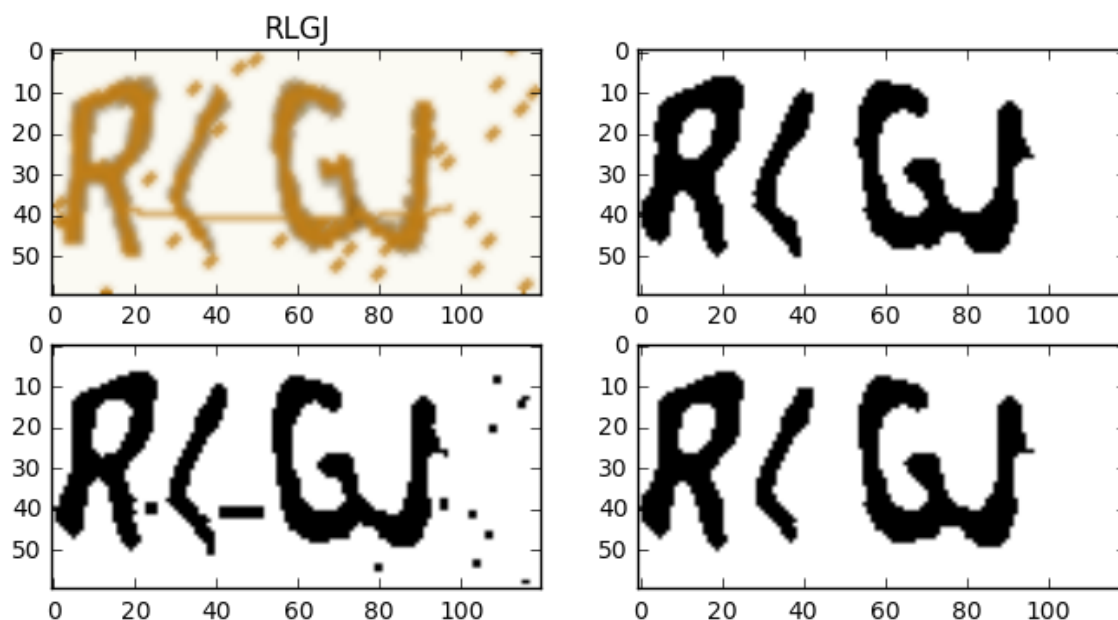
# 中值滤波
b1 = cv2.medianBlur(blur1, 5)
b2 = cv2.medianBlur(blur2, 5)
b3 = cv2.medianBlur(blur3, 5)
"""
# 均值滤波
b1 = cv2.blur(blur1, (5,5))
b2 = cv2.blur(blur2, (5,5))
b3 = cv2.blur(blur3, (5,5))
"""
# 高斯滤波
b1 = cv2.GaussianBlur(blur1, (5,5),0)
b2 = cv2.GaussianBlur(blur2, (5,5),0)
b3 = cv2.GaussianBlur(blur3, (5,5),0)
"""

plt.figure(figsize = (8, 4))
plt.subplot(2, 2, 1)
plt.imshow(img, cmap='gray')
plt.title(random_str)
plt.subplot(2, 2, 2)
plt.imshow(b1, cmap='gray')
plt.subplot(2, 2, 3)
plt.imshow(b2, cmap='gray')
plt.subplot(2, 2, 4)
plt.imshow(b3, cmap='gray')

```

Out[4]:

<matplotlib.image.AxesImage at 0x7fa47c36d400>



分析

初步分析，“二值化+滤波”方案比“形态学转换”方案的去噪效果更好，并且能够保留验证码原貌。然而多次测试发现，由于生成的验证码变化太大，单纯的“二值化+滤波”方案并不能很好地去噪点，强行去噪还会导致处理过的图片消失。

对于该版本的验证码本人没有研究出一种合适的方案，为了后续实验的简单，我们就不对验证码进行去噪操作。后续实验也可证明，该版本的验证码不去噪比去噪的破解率更高，然而这并不代表验证码的去噪操作无意义。

In [5]:

```

# 验证码生成器
def gen(batch_size=32):
    X = np.zeros((batch_size, height, width, 3), dtype=np.uint8)
    # X = np.zeros((batch_size, height, width, 1), 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)
            """
            # 去噪操作
            img = generator.generate_image(random_str)
            # 灰度图
            img = cv2.cvtColor(np.array(img), cv2.COLOR_BGR2GRAY)
            # Otsu's 二值化
            _, img = cv2.threshold(gray, 230, 255, cv2.THRESH_BINARY)
            # 中值滤波
            img = cv2.medianBlur(img, 5)
            X[i] = img.reshape((height, width, 1))
            """
            for j, ch in enumerate(random_str):
                y[j][i, :] = 0
                y[j][i, characters.find(ch)] = 1
        yield X, y

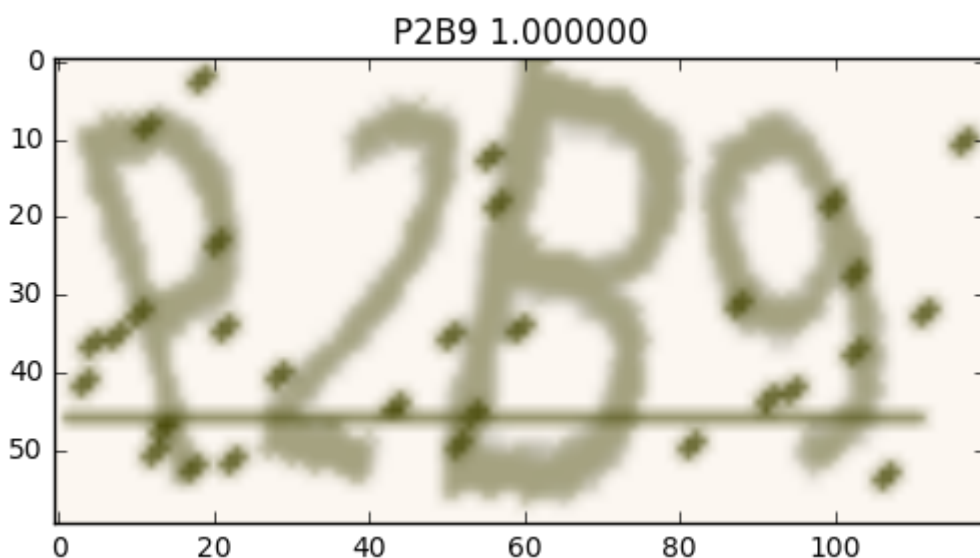
# 验证码识别器
def decode(y):
    acc = np.cumprod(np.max(y, axis=2)[0:])[0:-1]
    y = np.argmax(np.array(y), axis=2)[: ,0]
    title = ''.join([characters[x] for x in y])
    return '%s %.6f' % (title, acc)

# 测试效果
X, y = next(gen(1))
plt.imshow(X[0], cmap='gray')
plt.title(decode(y))

```

Out[5]:

<matplotlib.text.Text at 0x7fa47c2af9b0>



In [6]:

```
from keras.models import *
from keras.layers import *

## 定义模型 http://keras-cn.readthedocs.io/en/latest/models/model/
input_tensor = Input((height, width, 3))
x = input_tensor # (None, 60, 120, 3)
"""
# 每次经过两次3*3的卷积层和2*2的最大池化层 (conv3-32 - conv3-32 - maxpool)
for i in range(3):
    x = Conv2D(32*2**i, (3, 3), activation='relu')(x)
    x = ZeroPadding2D((1,1))(x)
    x = Conv2D(32*2**i, (3, 3), activation='relu')(x)
    x = ZeroPadding2D((1,1))(x)
    x = MaxPooling2D((2, 2))(x)
"""
## 等价于上面注释部分
x = Conv2D(32, (3, 3), activation='relu')(x) # (None, 58, 118, 32)
x = ZeroPadding2D((1,1))(x) # (None, 60, 120, 32)
x = Conv2D(32, (3, 3), activation='relu')(x) # (None, 58, 118, 32)
x = ZeroPadding2D((1,1))(x) # (None, 60, 120, 32)
x = MaxPooling2D((2, 2))(x) # (None, 30, 60, 32)

x = Conv2D(64, (3, 3), activation='relu')(x) # (None, 28, 58, 64)
x = ZeroPadding2D((1,1))(x) # (None, 30, 60, 64)
x = Conv2D(64, (3, 3), activation='relu')(x) # (None, 28, 58, 64)
x = ZeroPadding2D((1,1))(x) # (None, 30, 60, 64)
x = MaxPooling2D((2, 2))(x) # (None, 15, 30, 64)

x = Conv2D(128, (3, 3), activation='relu')(x) # (None, 13, 28, 128)
x = ZeroPadding2D((1,1))(x) # (None, 15, 30, 128)
x = Conv2D(128, (3, 3), activation='relu')(x) # (None, 13, 28, 128)
x = ZeroPadding2D((1,1))(x) # (None, 15, 30, 128)
x = MaxPooling2D((2, 2))(x) # (None, 7, 15, 128)

x = Flatten()(x)
x = Dropout(0.5)(x)
# 4个分类器多输出
x = [Dense(n_class, activation='softmax', name='c%d'%(i+1))(x) for i in range(4)]

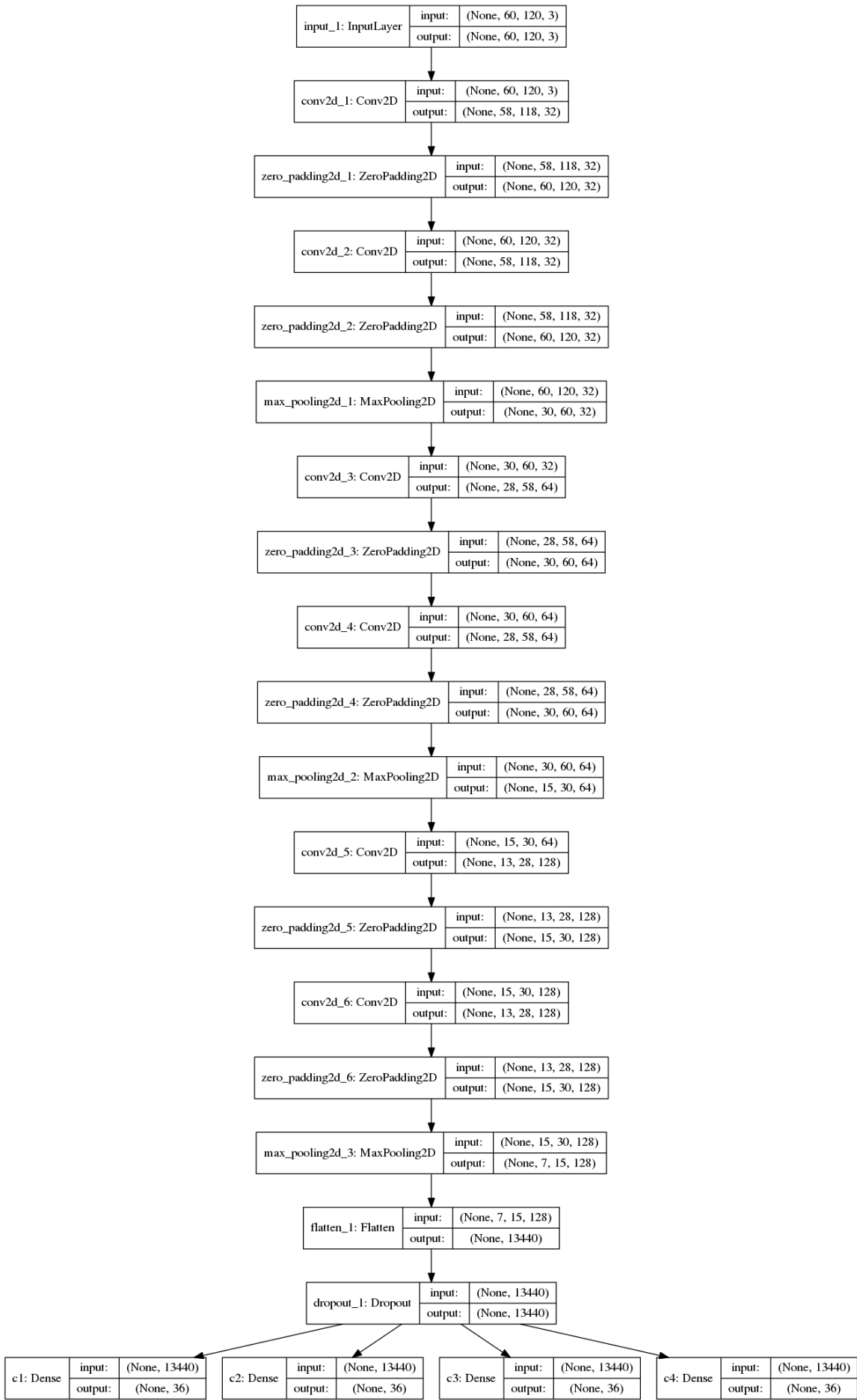
model = Model(inputs=input_tensor, outputs=x)
model.compile(loss='categorical_crossentropy',
              optimizer='adadelta',
              metrics=['accuracy'])
```

Using TensorFlow backend.

In [7]:

```
# 模型可视化 https://keras-cn.readthedocs.io/en/latest/other/visualization/  
from keras.utils import plot_model  
from IPython import display  
# sudo pip install graphviz(安装接口)  
# sudo apt-get install graphviz(安装软件本身)  
# sudo pip install pydot  
plot_model(model, to_file="model.png", show_shapes=True)  
display.Image('model.png')
```

Out[7]:



In [8]:

```
# 训练模型 https://keras-cn.readthedocs.io/en/latest/models/model/
# model.fit_generator(gen(), steps_per_epoch=1000, epochs=10, workers=4, use_multiprocessing=True,
#                       validation_data=gen(), validation_steps=200)
model.fit_generator(gen(), steps_per_epoch=1000, epochs=10, validation_data=gen(), validation_steps=200)
```

Epoch 1/10

1000/1000 [=====] - 1071s 1s/step - loss: 13.9193 - c1_loss: 3.2167 - c2_loss: 3.5594 - c3_loss: 3.5788 - c4_loss: 3.5643 - c1_acc: 0.1166 - c2_acc: 0.0494 - c3_acc: 0.0458 - c4_acc: 0.0477 - val_loss: 8.9915 - val_c1_loss: 1.5588 - val_c2_loss: 2.4117 - val_c3_loss: 2.5605 - val_c4_loss: 2.4605 - val_c1_acc: 0.5773 - val_c2_acc: 0.3444 - val_c3_acc: 0.3102 - val_c4_acc: 0.3387

Epoch 2/10

1000/1000 [=====] - 1093s 1s/step - loss: 5.1885 - c1_loss: 0.8434 - c2_loss: 1.4017 - c3_loss: 1.5958 - c4_loss: 1.3475 - c1_acc: 0.7199 - c2_acc: 0.5647 - c3_acc: 0.5159 - c4_acc: 0.5836 - val_loss: 1.5717 - val_c1_loss: 0.2205 - val_c2_loss: 0.3937 - val_c3_loss: 0.5266 - val_c4_loss: 0.4309 - val_c1_acc: 0.9295 - val_c2_acc: 0.8700 - val_c3_acc: 0.8317 - val_c4_acc: 0.8677

Epoch 3/10

1000/1000 [=====] - 1135s 1s/step - loss: 1.8753 - c1_loss: 0.2425 - c2_loss: 0.4944 - c3_loss: 0.6592 - c4_loss: 0.4791 - c1_acc: 0.9167 - c2_acc: 0.8358 - c3_acc: 0.7944 - c4_acc: 0.8426 - val_loss: 0.7255 - val_c1_loss: 0.0722 - val_c2_loss: 0.1977 - val_c3_loss: 0.2690 - val_c4_loss: 0.1865 - val_c1_acc: 0.9722 - val_c2_acc: 0.9291 - val_c3_acc: 0.9167 - val_c4_acc: 0.9355

Epoch 4/10

1000/1000 [=====] - 1221s 1s/step - loss: 1.1011 - c1_loss: 0.1367 - c2_loss: 0.2864 - c3_loss: 0.4002 - c4_loss: 0.2778 - c1_acc: 0.9534 - c2_acc: 0.9035 - c3_acc: 0.8739 - c4_acc: 0.9083 - val_loss: 0.5347 - val_c1_loss: 0.0666 - val_c2_loss: 0.1221 - val_c3_loss: 0.2138 - val_c4_loss: 0.1322 - val_c1_acc: 0.9731 - val_c2_acc: 0.9547 - val_c3_acc: 0.9355 - val_c4_acc: 0.9528

Epoch 5/10

1000/1000 [=====] - 1165s 1s/step - loss: 0.8221 - c1_loss: 0.0978 - c2_loss: 0.2026 - c3_loss: 0.3184 - c4_loss: 0.2032 - c1_acc: 0.9658 - c2_acc: 0.9321 - c3_acc: 0.9014 - c4_acc: 0.9331 - val_loss: 0.4322 - val_c1_loss: 0.0523 - val_c2_loss: 0.0968 - val_c3_loss: 0.1699 - val_c4_loss: 0.1132 - val_c1_acc: 0.9808 - val_c2_acc: 0.9627 - val_c3_acc: 0.9458 - val_c4_acc: 0.9592

Epoch 6/10

1000/1000 [=====] - 1412s 1s/step - loss: 0.6597 - c1_loss: 0.0865 - c2_loss: 0.1533 - c3_loss: 0.2552 - c4_loss: 0.1647 - c1_acc: 0.9680 - c2_acc: 0.9470 - c3_acc: 0.9203 - c4_acc: 0.9446 - val_loss: 0.3480 - val_c1_loss: 0.0512 - val_c2_loss: 0.0765 - val_c3_loss: 0.1438 - val_c4_loss: 0.0765 - val_c1_acc: 0.9788 - val_c2_acc: 0.9694 - val_c3_acc: 0.9544 - val_c4_acc: 0.9711

Epoch 7/10

1000/1000 [=====] - 1135s 1s/step - loss: 0.5584 - c1_loss: 0.0750 - c2_loss: 0.1287 - c3_loss: 0.2192 - c4_loss: 0.1355 - c1_acc: 0.9721 - c2_acc: 0.9537 - c3_acc: 0.9331 - c4_acc: 0.9531 - val_loss: 0.3463 - val_c1_loss: 0.0489 - val_c2_loss: 0.0689 - val_c3_loss: 0.1428 - val_c4_loss: 0.0857 - val_c1_acc: 0.9836 - val_c2_acc: 0.9747 - val_c3_acc: 0.9581 - val_c4_acc: 0.9705

Epoch 8/10

1000/1000 [=====] - 1073s 1s/step - loss: 0.5005 - c1_loss: 0.0652 - c2_loss: 0.1155 - c3_loss: 0.1954 - c4_loss: 0.1243 - c1_acc: 0.9751 - c2_acc: 0.9584 - c3_acc: 0.9399 - c4_acc: 0.9583 - val_loss: 0.2746 - val_c1_loss: 0.0519 - val_c2_loss: 0.0576 - val_c3_loss: 0.1071 - val_c4_loss: 0.0581 - val_c1_acc: 0.9778 - val_c2_acc: 0.9769 - val_c3_acc: 0.9648 - val_c4_acc: 0.9780

Epoch 9/10

1000/1000 [=====] - 1077s 1s/step - loss: 0.4547 - c1_loss: 0.0623 - c2_loss: 0.1046 - c3_loss: 0.1758 - c4_loss: 0.1121 - c1_acc: 0.9758 - c2_acc: 0.9628 - c3_acc: 0.9443 - c4_acc: 0.9627 - val_loss: 0.3106 - val_c1_loss: 0.0466 - val_c2_loss:

```
0.0718 - val_c3_loss: 0.1183 - val_c4_loss: 0.0739 - val_c1_acc: 0.9
819 - val_c2_acc: 0.9734 - val_c3_acc: 0.9633 - val_c4_acc: 0.9727
Epoch 10/10
1000/1000 [=====] - 1080s 1s/step - loss:
0.4073 - c1_loss: 0.0567 - c2_loss: 0.0939 - c3_loss: 0.1581 - c4_lo
ss: 0.0986 - c1_acc: 0.9781 - c2_acc: 0.9654 - c3_acc: 0.9503 - c4_a
cc: 0.9649 - val_loss: 0.2355 - val_c1_loss: 0.0405 - val_c2_loss:
0.0455 - val_c3_loss: 0.0970 - val_c4_loss: 0.0525 - val_c1_acc: 0.9
831 - val_c2_acc: 0.9814 - val_c3_acc: 0.9689 - val_c4_acc: 0.9784
```

Out[8]:

<keras.callbacks.History at 0x7fa44e208dd8>

In [9]:

```
# 保存模型
model.save('captcha_model.h5')
```

In [10]:

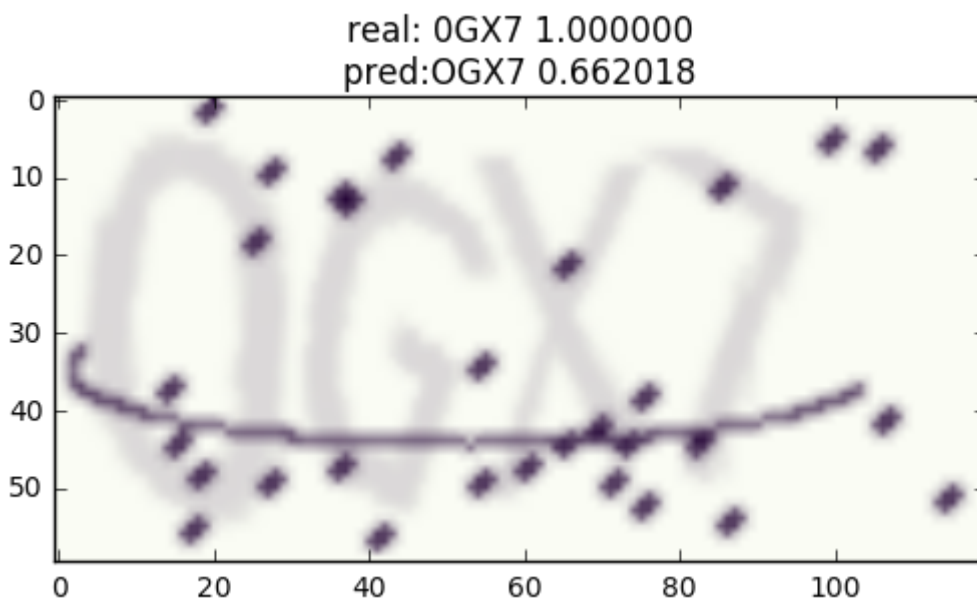
```
# 加载模型
from keras.models import load_model
model = load_model('captcha_model.h5')
```

In [11]:

```
# 预测效果
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')
```

Out[11]:

<matplotlib.image.AxesImage at 0x7fa44c7256a0>



In [12]:

```

# 评估效果
from tqdm import tqdm # https://pypi.python.org/pypi/tqdm
def evaluate(model, batch_num=100):
    batch_acc = 0
    for i in tqdm(range(batch_num)):
        X, y = next(gen(1))
        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(list(map(np.array_equal, y_true, y_pred))) # python3
        # mean 改 amin 保证4个字符全部正确
        batch_acc += np.amin(list(map(np.array_equal, y_true, y_pred))) # python3
    return batch_acc / batch_num
evaluate(model)

```

```
100%|██████████| 100/100 [00:01<00:00, 54.12it/s]
```

Out[12]:

0.93999999999999995

分析

利用深度学习识别验证码可以达到80-90%的破解率，两次破解都失败的几率仅为4%-9%，这点失败率在允许错误次数面前显得微不足道。

因此，可以大胆地下结论：目前的图形验证码还不够安全。

对抗样本

什么是对抗样本

Christian Szegedy等人在ICLR2014发表的论文中，他们提出了对抗样本（Adversarial examples）的概念，即在数据集中通过故意添加细微的干扰所形成的输入样本，受干扰之后的输入导致模型以高置信度给出一个错误的输出。在他们的论文中，他们发现包括卷积神经网络（Convolutional Neural Network, CNN）在内的深度学习模型对于对抗样本都具有极高的脆弱性。他们的研究提到，很多情况下，在训练集的不同子集上训练得到的具有不同结构的模型都会对相同的对抗样本实现误分，这意味着对抗样本成为了训练算法的一个盲点。



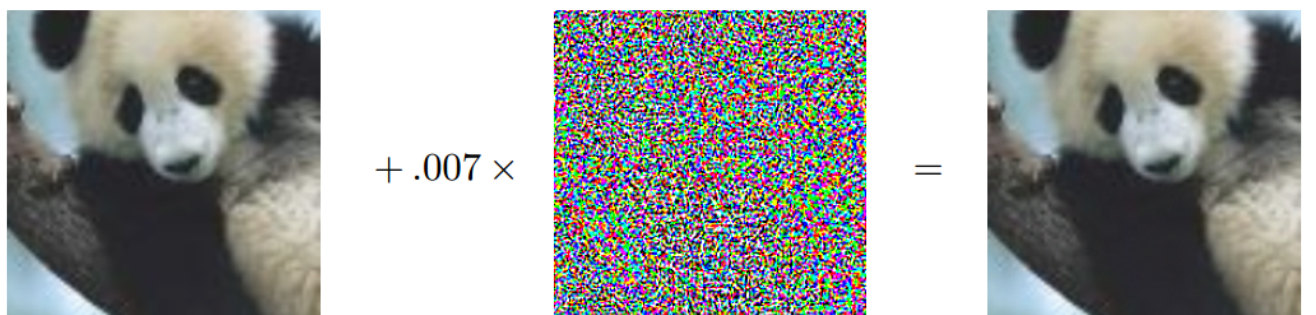
Figure 5: Adversarial examples generated for AlexNet [9].(Left) is a correctly predicted sample, (center) difference between correct image, and image predicted incorrectly magnified by 10x (values shifted by 128 and clamped), (right) adversarial example. All images in the right column are predicted to be an “ostrich, *Struthio camelus*”. Average distortion based on 64 examples is 0.006508. Please refer to <http://goo.gl/huaGPb> for full resolution images. The examples are strictly randomly chosen. There is not any postselection involved.

如何产生对抗样本

生成式对抗网络的发明人Ian Goodfellow在《Explaining and Harnessing Adversarial Examples》中提出了一种更快速方便的方法来产生对抗样本：

$$X' = X + \epsilon \cdot \text{sign}(\nabla_x J(\theta, x, y))$$

这种方法的思想非常简单，就是让输入图像朝着让类别置信度降低的方向上移动一个在各个维度上都是 ϵ 这么大小的一步。因为输入通常是高维的（比如224x224），再加上现在的主流神经网络结构都是ReLU系的激活函数，线性程度其实很高，所以即使是很小的 ϵ ，每个维度的效果加一块，通常也足以对结果产生很大的影响。在计算上，这种方法优势巨大，因为只需要一次前向和一次后向梯度计算就可以了。Ian Goodfellow称之为Fast Gradient Sign method。



$$x + .007 \times \text{sign}(\nabla_x J(\theta, x, y)) = x + \epsilon \text{sign}(\nabla_x J(\theta, x, y))$$

“panda”
57.7% confidence

“nematode”
8.2% confidence

“gibbon”
99.3 % confidence

深度学习的脆弱性

一个推断性的解释是深度神经网络的高度非线性特征，以及纯粹的监督学习模型中不充分的模型平均和不充分的正则化所导致的过拟合。Ian Goodfellow 在ICLR2015年的论文中，通过在一个线性模型加入对抗干扰，发现只要线性模型的输入拥有足够的维度（事实上大部分情况下，模型输入的维度都比较大，因为维度过小的输入会导致模型的准确率过低），线性模型也对对抗样本表现出明显的脆弱性，这也驳斥了关于对抗样本是因为模型的高度非线性的解释。相反深度学习的对抗样本是由于模型的线性特征。

对抗样本的危害性

如果我们有两张图片，人眼看上去一模一样，都是一间房子，但是cnn把一张分类为房子，一张分类为鸵鸟，这种分类器还有什么用啊。如果我们的模型随随便便就被欺骗，还能被投入使用吗，那那些研究不成了纸上谈兵了吗。另一方面，有人觉得即使是Ian的方法，都需要知道梯度才能找到对抗样本，可是攻击者怎么知道模型是什么样子呢？可惜对抗样本还有一个重要性质，叫做Transferability，转移性。很多的时候，两个模型即使有不同的结构并在不同的训练集上被训练，一种模型的对抗样本在另一个模型中也同样会被误分，甚至它们还会将对抗样本误分为相同的类。这是因为对抗样本与模型的权值向量高度吻合，同时为了训练执行相同的任务，不同的模型学习了相似的函数。这种泛化特征意味着如果有人希望对模型进行恶意攻击，攻击者根本不必访问需要攻击的目标模型，就可以通过训练自己的模型来产生对抗样本，然后将这些对抗样本部署到他们需要攻击的模型中。这个可以称作灾难了，最后的堡垒也被攻破了，我们再怎么隐藏模型的细节，也可能被攻击。

换个思路走的更远

正如攻击者可以利用对抗本来攻击深度学习模型，同理我们也可以利用对抗本来巩固自己的深度学习模型，例如前面已经被破解的验证码.....

参考论文

1. Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, Rob Fergus - [Intriguing properties of neural networks](https://arxiv.org/abs/1312.6199) (<https://arxiv.org/abs/1312.6199>)
2. Ian J. Goodfellow, Jonathon Shlens, Christian Szegedy - [Explaining and Harnessing Adversarial Examples](https://arxiv.org/abs/1412.6572) (<https://arxiv.org/abs/1412.6572>)
3. Karen Simonyan, Andrew Zisserman - [Very Deep Convolutional Networks for Large-Scale Image Recognition](https://arxiv.org/abs/1409.1556) (<https://arxiv.org/abs/1409.1556>)

参考链接

1. <http://www.infoq.com/cn/news/2015/07/adversarial-examples> (<http://www.infoq.com/cn/news/2015/07/adversarial-examples>)
2. <http://blog.csdn.net/cdpac/article/details/53170940> (<http://blog.csdn.net/cdpac/article/details/53170940>)
3. <https://zhuanlan.zhihu.com/p/26122612> (<https://zhuanlan.zhihu.com/p/26122612>)

阶段二：验证码对抗

参考链接

1. https://github.com/rodgzilla/machine_learning_adversarial_examples (https://github.com/rodgzilla/machine_learning_adversarial_examples)

In [13]:

```

from keras import metrics
from keras.utils.np_utils import to_categorical
import keras.backend as K

# 生成对抗样本
def generate_adversarial(model, x):
    original = np.array(x)
    target_idx = np.argmax(model.predict(original), axis=2)[:,-1]
    target = to_categorical(target_idx, n_class)
    target_variable = K.variable(target)
    loss = metrics.categorical_crossentropy(model.output, target_variable)
    gradients = K.gradients(loss, model.input)
    # Keras训练和测试两种模式下不完全一致
    # 测试模式忽略Dropout层、BN层等组件
    # 需要在函数中传递一个learning_phase的标记 0训练/1测试
    get_grad_values = K.function([model.input, K.learning_phase()], gradients)
    grad_values = get_grad_values([original, 0])[0]

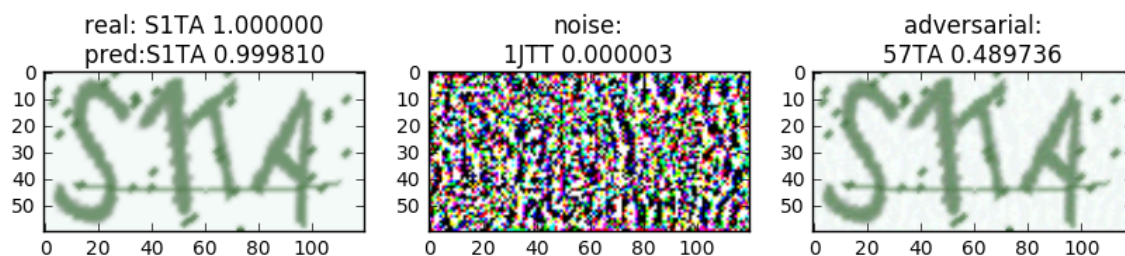
    grad_signs = np.sign(grad_values) # 取符号-1 +1 0
    epsilon = 4 # 模糊因子
    noise = grad_signs * epsilon # 噪声
    adversarial = np.clip(original + noise, 0., 255.).astype(np.uint8) # 取值(0,255)
    return original, noise, adversarial

# 展示对比效果
def show(model, original, noise, adversarial):
    ##### 原始样本
    title_original = 'real: %s\npred:%s' % (decode(y), decode(model.predict(original)))
    ##### 噪声
    title_noise = 'noise:\n%s' % decode(model.predict(noise))
    ##### 对抗样本
    title_adversarial = 'adversarial:\n%s' % decode(model.predict(adversarial))

    plt.figure(figsize = (10, 5))
    plt.subplot(1, 3, 1)
    plt.imshow(original[0], cmap='gray')
    plt.title(title_original)
    plt.subplot(1, 3, 2)
    plt.imshow(noise[0], cmap='gray')
    plt.title(title_noise)
    plt.subplot(1, 3, 3)
    plt.imshow(adversarial[0], cmap='gray')
    plt.title(title_adversarial)

X, y = next(gen(1))
original, noise, adversarial = generate_adversarial(model, X)
show(model, original, noise, adversarial)

```

反对抗样本

众所周知，在图像识别领域，对抗样本是一个非常棘手的问题，研究如何克服它们可以帮助避免潜在的危险。虽然对抗样本看似可怕，但是只要对抗样本自身发生轻微变化，对抗性质也会失灵。

参考链接

1. <https://www.leiphone.com/news/201707/8vQdkhyfsSf0PkBP.html>
(<https://www.leiphone.com/news/201707/8vQdkhyfsSf0PkBP.html>)
2. <https://www.leiphone.com/news/201707/U58CXHRIgixDYd98.html>
(<https://www.leiphone.com/news/201707/U58CXHRIgixDYd98.html>)

阶段三：验证码反对抗

参考链接

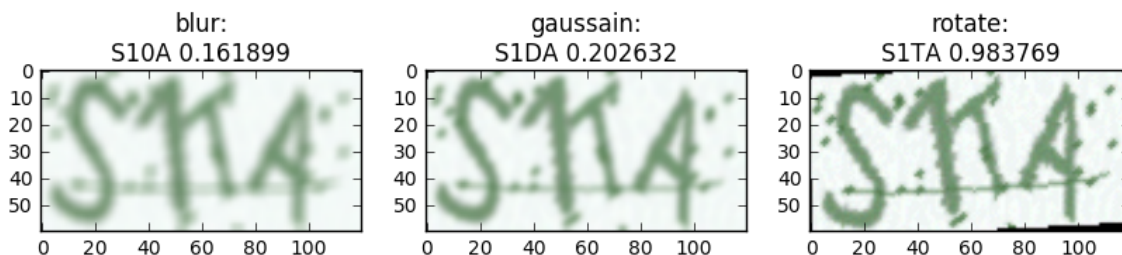
1. https://github.com/rodgzilla/machine_learning_adversarial_example
(https://github.com/rodgzilla/machine_learning_adversarial_example)

In [14]:

```
# 生成反对抗样本
def generate_anti_adversarial(adversarial):
    ##### 均值滤波
    img_blur = cv2.blur(adversarial[0], (5,5))
    title_blur = 'blur:\n%s' % decode(model.predict(np.expand_dims(img_blur,
0)))
    ##### 高斯滤波
    img_gaussain = cv2.GaussianBlur(adversarial[0], (5,5), 0)
    title_gaussain = 'gaussain:\n%s' % decode(model.predict(np.expand_dims(img_g
aussain, 0)))
    ##### 旋转对抗
    img_rotate = Image.fromarray(adversarial[0]).rotate(3)
    title_rotate = 'rotate:\n%s' % decode(model.predict(np.expand_dims(img_rot
ate, 0)))

    plt.figure(figsize = (10, 5))
    plt.subplot(1, 3, 1)
    plt.imshow(img_blur, cmap='gray')
    plt.title(title_blur)
    plt.subplot(1, 3, 2)
    plt.imshow(img_gaussain, cmap='gray')
    plt.title(title_gaussain)
    plt.subplot(1, 3, 3)
    plt.imshow(img_rotate, cmap='gray')
    plt.title(title_rotate)

generate_anti_adversarial(adversarial)
```



阶段四：实验对比

接下来我们来做几组对比：

1. 原始样本
2. 对抗样本
3. 反对抗样本
 - A. 均值滤波
 - B. 高斯滤波
 - C. 旋转对抗

In [15]:

```
import os

# 从文件夹加载验证码
# 对比实验控制变量法
def load_data(data_path):
    flist = os.listdir(test_path)
    flen = len(flist)
    X = np.zeros((flen, height, width, 3), dtype=np.uint8)
    y = np.zeros((flen, n_len), dtype=np.uint8)
    for i in range(flen):
        X[i] = Image.open(test_path + flist[i])
        for j, ch in enumerate(flist[i][0:4]):
            y[i][j] = characters.find(ch) # array([ 3,  8, 15, 11], dtype=uint8)
    return X, y, len(flist)

test_path = 'test/'
X, y, flen= load_data(test_path)
```

In [16]:

```
# 原始样本
acc = 0
# for i in tqdm(range(flen)):
for i in tqdm(range(100)):
    y_pred = model.predict(np.expand_dims(X[i], 0)) # (60, 120, 3)扩展到(60, 120, 3, 1)
    y_pred = np.argmax(y_pred, axis=2).T
    acc += np.amin(list(map(np.array_equal, y[i], y_pred[0]))) # python3
# acc / flen
acc / 100
```

```
100%|██████████| 100/100 [00:01<00:00, 68.40it/s]
```

Out[16]:

```
0.92000000000000004
```

In [17]:

```
# 对抗样本
acc = 0
for i in tqdm(range(100)):
    x = np.expand_dims(X[i], 0)
    _, _, adversarial = generate_adversarial(model, x) # 生成对抗样本
    y_pred = model.predict(adversarial) # (60, 120, 3)扩展到(60, 120, 3, 1)
    y_pred = np.argmax(y_pred, axis=2).T
    acc += np.amin(list(map(np.array_equal, y[i], y_pred[0]))) # python3
acc / 100
```

```
100%|██████████| 100/100 [01:40<00:00, 1.00s/it]
```

Out[17]:

```
0.059999999999999998
```

In [18]:

```
# 反对抗样本-均值滤波
acc = 0
for i in tqdm(range(100)):
    x = np.expand_dims(X[i], 0)
    _, _, adversarial = generate_adversarial(model, x) # 生成对抗样本
    img_blur = cv2.blur(adversarial[0], (5,5)) # 生成反对抗样本
    y_pred = model.predict(np.expand_dims(img_blur, 0)) # (60, 120, 3)扩展到(60, 120, 3, 1)
    y_pred = np.argmax(y_pred, axis=2).T
    acc += np.amin(list(map(np.array_equal, y[i], y_pred[0]))) # python3
acc / 100
```

```
100%|██████████| 100/100 [03:40<00:00, 2.20s/it]
```

Out[18]:

0.32000000000000001

In [19]:

```
# 反对抗样本-高斯滤波
acc = 0
for i in tqdm(range(100)):
    x = np.expand_dims(X[i], 0)
    _, _, adversarial = generate_adversarial(model, x) # 生成对抗样本
    img_gaussian = cv2.GaussianBlur(adversarial[0], (5,5), 0) # 生成反对抗样本
    y_pred = model.predict(np.expand_dims(img_gaussian, 0)) # (60, 120, 3)扩展到(60, 120, 3, 1)
    y_pred = np.argmax(y_pred, axis=2).T
    acc += np.amin(list(map(np.array_equal, y[i], y_pred[0]))) # python3
acc / 100
```

```
100%|██████████| 100/100 [05:38<00:00, 3.39s/it]
```

Out[19]:

0.46000000000000002

In [20]:

```
# 反对抗样本-旋转对抗
acc = 0
for i in tqdm(range(100)):
    x = np.expand_dims(X[i], 0)
    _, _, adversarial = generate_adversarial(model, x) # 生成对抗样本
    img_rot = Image.fromarray(adversarial[0]).rotate(3) # 生成反对抗样本
    y_pred = model.predict(np.expand_dims(img_rot, 0)) # (60, 120, 3)扩展到(60, 120, 3, 1)
    y_pred = np.argmax(y_pred, axis=2).T
    acc += np.amin(list(map(np.array_equal, y[i], y_pred[0]))) # python3
acc / 100
```

```
100%|██████████| 100/100 [07:42<00:00, 4.63s/it]
```

Out[20]:

0.42999999999999999

分析

从这5组实验结果可以看出，虽然对抗样本迷惑了最初的深度学习模型，但是对抗样本在进行滤波和旋转等处理后，识别率还能恢复到三分之一左右。

因此，利用对抗本来防御恶意用户可能不是一个有效的方法，我们应该从其他方面改进验证码。

当然如何提高对抗样本的健壮性也可以是我们未来研究的目标之一。比如利用开源对抗学习库CleverHans(<https://github.com/tensorflow/cleverhans>)，可以开发更加健壮的机器学习深度学习模型。

验证码的挑战

在上世纪五十年代，人工智能之父阿兰·图灵设计出了图灵测试。在约半个世纪后，图灵测试的理念被做成最简单粗暴的形式——验证码，渗透到人们互联网生活的方方面面。然而，验证码是一个时代的产物，是一种治标不治本的速效手段。战术和战略的改变，才是终极解决方法。

对抗与反对抗之路上诞生了形形色色的验证码：

1. 基于视觉推理
2. 基于购买记录
3. 滑块验证码
4. 拼图验证码
5.

可以预想，终有一天验证码会退出互联网的历史舞台。但现阶段，由于巨额潜在利润的驱动，不法之徒必定不会放弃对验证码的虎视眈眈。无论是过去、现在，亦或是不远的将来，这都注定是一场没有硝烟的血战。未来，我们拭目以待。