Python+OpenCV實現自動掃雷，創造屬於自己的世界記錄！

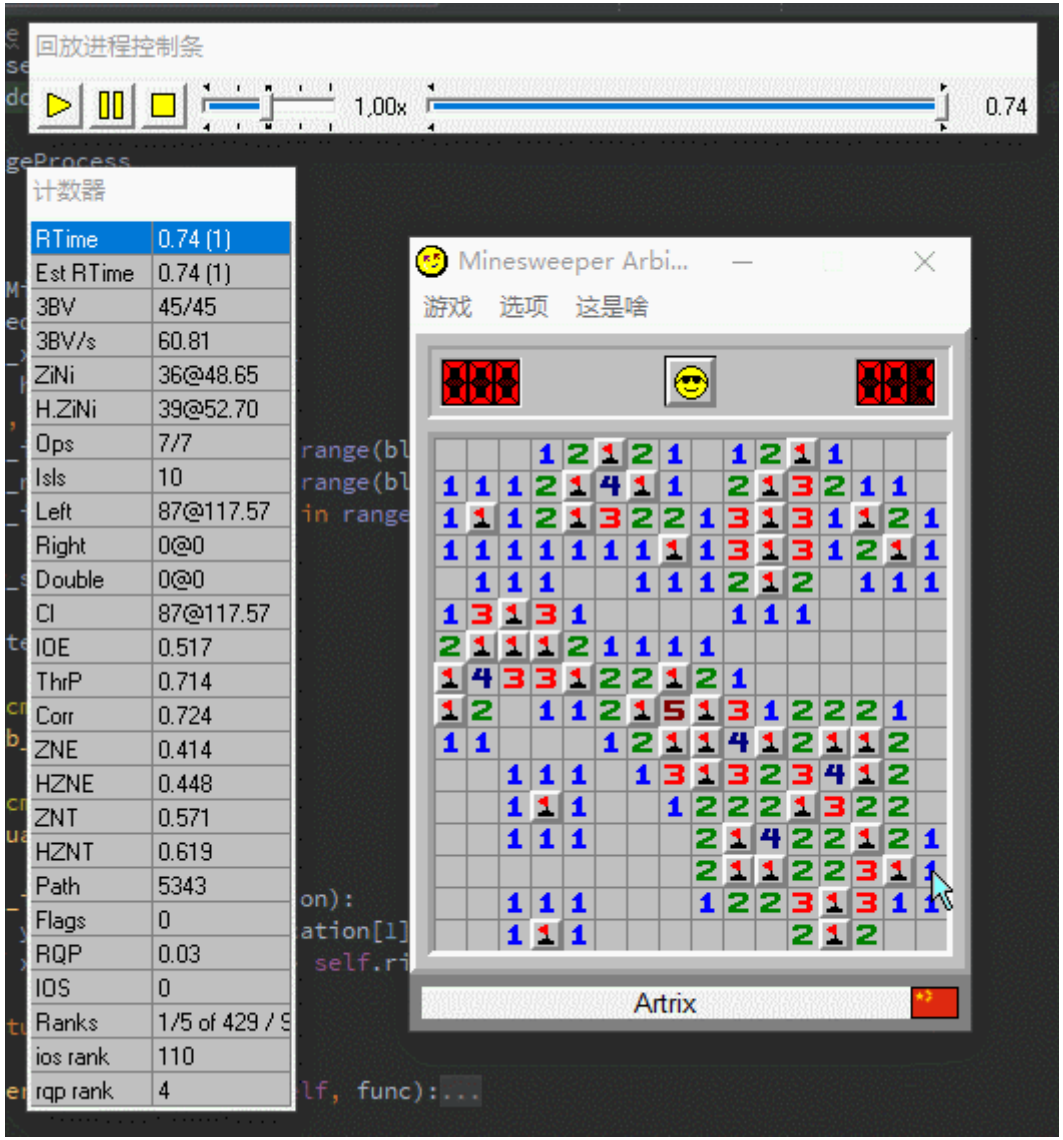
小白 小白學視覺 今天

點擊上方

重磅乾貨，第一時間送達

> **本文轉載自知乎Artrix**
> **https://zhuanlan.zhihu.com/p/35755039**

五一勞動節假期，我們一起來玩掃雷吧。用Python+OpenCV實現了自動掃雷，突破世界記錄，我們先來看一下效果吧。



中級- 0.74秒3BV/S=60.81

相信許多人很早就知道有掃雷這麼一款經典的遊（顯卡測試）戲（軟件），更是有不少人曾聽說過中國雷聖，也是中國掃雷第一、世界綜合排名第二的郭蔚嘉的頂頂大名。掃雷作為一款在Windows9x時代就已經誕生的經典遊戲，從過去到現在依然都有著它獨特的魅力：快節奏高精準的鼠標操作要求、快速的反應能力、刷新紀錄的快感，這些都是掃雷給雷友們帶來的、只屬於掃雷的獨一無二的興奮點。

準備動手製作一套掃雷自動化軟件之前，你需要準備如下一些工具/軟件/環境

- 開發環境

1. Python3 環境- 推薦3.6或者以上
2. numpy依賴庫[如有Anaconda則無需安裝]
3. PIL依賴庫[如有Anaconda則無需安裝]
4. opencv-python
5. win32gui、win32api依賴庫
6. 支持Python的IDE [可選，如果你能忍受用文本編輯器寫程序也可以]

- 掃雷軟件

- **Minesweeper Arbiter 下載地址（必須使用MS-Arbiter來進行掃雷！）**

好啦，那麼我們的準備工作已經全部完成了！讓我們開始吧~

在去做一件事情之前最重要的是什麼？是將要做的這件事情在心中搭建一個步驟框架。只有這樣，才能保證在去做這件事的過程中，盡可能的做到深思熟慮，使得最終有個好的結果。我們寫程序也要盡可能做到在正式開始開發之前，在心中有個大致的思路。

對於本項目而言，大致的開發過程是這樣的：

1. 完成窗體內容截取部分

2. 完成雷塊分割部分
3. 完成雷塊類型識別部分
4. 完成掃雷算法

好啦，既然我們有了個思路，那就擼起袖子大力干！

- 01 窗體截取

其實對於本項目而言，窗體截取是一個邏輯上簡單，實現起來卻相當麻煩的部分，而且還是必不可少的部分。我們通過Spy++得到了以下兩點信息：

```
class_name = "TMain"
title_name = "Minesweeper Arbiter "
```

- ms_arbiter.exe的主窗體類別為"TMain"
- ms_arbiter.exe的主窗體名稱為"Minesweeper Arbiter "

注意到了麼？主窗體的名稱後面有個空格。正是這個空格讓筆者困擾了一會兒，只有加上這個空格，win32gui才能夠正常的獲取到窗體的句柄。

本項目採用了win32gui來獲取窗體的位置信息，具體代碼如下：

```
hwnd = win32gui.FindWindow(class_name, title_name)
if hwnd:
left, top, right, bottom = win32gui.GetWindowRect(hwnd)
```

通過以上代碼，我們得到了窗體相對於整塊屏幕的位置。之後我們需要通過PIL來進行掃雷界面的棋盤截取。

我們需要先導入PIL庫

```
from PIL import ImageGrab
```

然後進行具體的操作。

```
left += 15
top += 101
right -= 15
bottom -= 43

rect = (left, top, right, bottom)
img = ImageGrab.grab().crop(rect)
```

聰明的你肯定一眼就發現了那些奇奇怪怪的Magic Numbers，沒錯，這的確是Magic Numbers，是我們通過一點點細微調節得到的整個棋盤相對於窗體的位置。

注意：這些數據僅在Windows10下測試通過，如果在別的Windows系統下，不保證相對位置的正確性，因為老版本的系統可能有不同寬度的窗體邊框。

橙色的區域是我們所需要的

好啦，棋盤的圖像我們有了，下一步就是對各個雷塊進行圖像分割了~

- 02 雷塊分割

在進行雷塊分割之前，我們事先需要了解雷塊的尺寸以及它的邊框大小。經過筆者的測量，在ms_arbiter下，每一個雷塊的尺寸為16px*16px。

知道了雷塊的尺寸，我們就可以進行每一個雷塊的裁剪了。首先我們需要知道在橫和豎兩個方向上雷塊的數量。

```
block_width, block_height = 16, 16
  blocks_x = int((right - left) / block_width)
  blocks_y = int((bottom - top) / block_height)
```

之後，我們建立一個二維數組用於存儲每一個雷塊的圖像，並且進行圖像分割，保存在之前建立的數組中。

```
def crop_block(hole_img, x, y):
        x1, y1 = x * block_width, y * block_height
        x2, y2 = x1 + block_width, y1 + block_height
return hole_img.crop((x1, y1, x2, y2))

blocks_img = [[0 for i in range(blocks_y)] for i in range(blocks_x)]

for y in range(blocks_y):
for x in range(blocks_x):
        blocks_img[x][y] = crop_block(img, x, y)
```

將整個圖像獲取、分割的部分封裝成一個庫，隨時調用就OK啦~在筆者的實現中，我們將這一部分封裝成了imageProcess.py，其中函數get_frame()用於完成上述的圖像獲取、分割過程。

- 03 雷塊識別

這一部分可能是整 筆者在進行雷塊檢測的時候採用了比較簡單的特徵，高效並且可以滿足要求。

```
def analyze_block(self, block, location):
```

```
    block = imageProcess.pil_to_cv(block)

    block_color = block[8, 8]
    x, y = location[0], location[1]

    # -1:Not opened
    # -2:Opened but blank
    # -3:Un initialized

    # Opened
if self.equal(block_color, self.rgb_to_bgr((192, 192, 192))):
if not self.equal(block[8, 1], self.rgb_to_bgr((255, 255, 255))):
self.blocks_num[x][y] = -2
self.is_started = True
else:
self.blocks_num[x][y] = -1

    elif self.equal(block_color, self.rgb_to_bgr((0, 0, 255))):
self.blocks_num[x][y] = 1

    elif self.equal(block_color, self.rgb_to_bgr((0, 128, 0))):
self.blocks_num[x][y] = 2

    elif self.equal(block_color, self.rgb_to_bgr((255, 0, 0))):
self.blocks_num[x][y] = 3

```

```
    elif self.equal(block_color, self.rgb_to_bgr((0, 0, 128))):
self.blocks_num[x][y] = 4

    elif self.equal(block_color, self.rgb_to_bgr((128, 0, 0))):
self.blocks_num[x][y] = 5

    elif self.equal(block_color, self.rgb_to_bgr((0, 128, 128))):
self.blocks_num[x][y] = 6

    elif self.equal(block_color, self.rgb_to_bgr((0, 0, 0))):
if self.equal(block[6, 6], self.rgb_to_bgr((255, 255, 255))):
            # Is mine
self.blocks_num[x][y] = 9
        elif self.equal(block[5, 8], self.rgb_to_bgr((255, 0, 0))):
            # Is flag
self.blocks_num[x][y] = 0
else:
self.blocks_num[x][y] = 7

    elif self.equal(block_color, self.rgb_to_bgr((128, 128, 128))):
self.blocks_num[x][y] = 8
else:
self.blocks_num[x][y] = -3
self.is_mine_form = False

if self.blocks_num[x][y] == -3 or not self.blocks_num[x][y] == -1:
```

```
    self.is_new_start = False
```

可以看到，我們採用了讀取每個雷塊的中心點像素的方式來判斷雷塊的類別，並且針對插旗、未點開、已點開但是空白等情況進行了進一步判斷。具體色值是筆者直接取色得到的，並且屏幕截圖的色彩也沒有經過壓縮，所以通過中心像素結合其他特徵點來判斷類別已經足夠了，並且做到了高效率。

在本項目中，我們實現的時候採用瞭如下標註方式：

- 1-8：表示數字1到8
- 9：表示是地雷
- 0：表示插旗
- -1：表示未打開
- -2：表示打開但是空白
- -3：表示不是掃雷遊戲中的任何方塊類型

通過這種簡單快速又有效的方式，我們成功實現了高效率的圖像識別。

- 04 掃雷算法實現

這可能是本篇文章最激動人心的部分了。在這裡我們需要先說明一下具體的掃雷算法思路：

1. 遍歷每一個已經有數字的雷塊，判斷在它周圍的九宮格內未被打開的雷塊數量是否和本身數字相同，如果相同則表明周圍九宮格內全部都是地雷，進行標記。
2. 再次遍歷每一個有數字的雷塊，取九宮格範圍內所有未被打開的雷塊，去除已經被上一次遍歷標記為地雷的雷塊，記錄並且點開。
3. 如果以上方式無法繼續進行，那麼說明遇到了死局，選擇在當前所有未打開的雷塊中隨機點擊。（當然這個方法不是最優的，有更加優秀的解決方案，但是實現相對麻煩）

基本的掃雷流程就是這樣，那麼讓我們來親手實現它吧~

首先我們需要一個能夠找出一個雷塊的九宮格範圍的所有方塊位置的方法。因為掃雷遊戲的特殊性，在棋盤的四邊是沒有九宮格的邊緣部分的，所以我們需要篩選來排除掉可能超過邊界的訪問。

```
def generate_kernel(k, k_width, k_height, block_location):

     ls = []
     loc_x, loc_y = block_location[0], block_location[1]

for now_y in range(k_height):
for now_x in range(k_width):
if k[now_y][now_x]:
                rel_x, rel_y = now_x - 1, now_y - 1
                ls.append((loc_y + rel_y, loc_x + rel_x))
return ls

 kernel_width, kernel_height = 3, 3

# Kernel mode:[Row][Col]
 kernel = [[1, 1, 1], [1, 1, 1], [1, 1, 1]]

# Left border
if x == 0:
for i in range(kernel_height):
         kernel[i][0] = 0

# Right border
```

```
if x == self.blocks_x - 1:
for i in range(kernel_height):
        kernel[i][kernel_width - 1] = 0

# Top border
if y == 0:
for i in range(kernel_width):
        kernel[0][i] = 0

# Bottom border
if y == self.blocks_y - 1:
for i in range(kernel_width):
        kernel[kernel_height - 1][i] = 0

# Generate the search map
 to_visit = generate_kernel(kernel, kernel_width, kernel_height, location)
```

我們在這一部分通過檢測當前雷塊是否在棋盤的各個邊緣來進行核的刪除（在核中，1為保留，0為捨棄），之後通過generate_kernel函數來進行最終坐標的生成。

```
def count_unopen_blocks(blocks):
    count = 0
for single_block in blocks:
if self.blocks_num[single_block[1]][single_block[0]] == -1:
            count += 1
return count
```

```
7
8  def mark_as_mine(blocks):
9  for single_block in blocks:
10 if self.blocks_num[single_block[1]][single_block[0]] == -1:
11 self.blocks_is_mine[single_block[1]][single_block[0]] = 1
12
13 unopen_blocks = count_unopen_blocks(to_visit)
14 if unopen_blocks == self.blocks_num[x][y]:
15      mark_as_mine(to_visit)
```

在完成核的生成之後，我們有了一個需要去檢測的雷塊“地址簿”：to_visit。之後，我們通過count_unopen_blocks函數來統計周圍九宮格範圍的未打開數量，並且和當前雷塊的數字進行比對，如果相等則將所有九宮格內雷塊通過mark_as_mine函數來標註為地雷。

```
1  def mark_to_click_block(blocks):
2  for single_block in blocks:
3
4  # Not Mine
5  if not self.blocks_is_mine[single_block[1]][single_block[0]] == 1:
6  # Click-able
7  if self.blocks_num[single_block[1]][single_block[0]] == -1:
8
9  # Source Syntax: [y][x] - Converted
10 if not (single_block[1], single_block[0]) in self.next_steps:
11 self.next_steps.append((single_block[1], single_block[0]))
12
13 def count_mines(blocks):
```

```
    count = 0
for single_block in blocks:
if self.blocks_is_mine[single_block[1]][single_block[0]] == 1:
            count += 1
return count

mines_count = count_mines(to_visit)

if mines_count == block:
    mark_to_click_block(to_visit)
```

掃雷流程中的第二步我們也採用了和第一步相近的方法來實現。先用和第一步完全一樣的方法來生成需要訪問的雷塊的核，之後生成具體的雷塊位置，通過count_mines函數來獲取九宮格範圍內所有雷塊的數量，並且判斷當前九宮格內所有雷塊是否已經被檢測出來。

如果是，則通過mark_to_click_block函數來排除九宮格內已經被標記為地雷的雷塊，並且將剩餘的安全雷塊加入next_steps數組內。

```
# Analyze the number of blocks
self.iterate_blocks_image(BoomMine.analyze_block)

# Mark all mines
self.iterate_blocks_number(BoomMine.detect_mine)

# Calculate where to click
self.iterate_blocks_number(BoomMine.detect_to_click_block)

if self.is_in_form(mouseOperation.get_mouse_point()):
```

```
for to_click in self.next_steps:
        on_screen_location = self.rel_loc_to_real(to_click)
        mouseOperation.mouse_move(on_screen_location[0], on_screen_location[1])
        mouseOperation.mouse_click()
```

在最終的實現內，筆者將幾個過程都封裝成為了函數，並且可以通過iterate_blocks_number方法來對所有雷塊都使用傳入的函數來進行處理，這有點類似Python中Filter的作用。

之後筆者做的工作就是判斷當前鼠標位置是否在棋盤之內，如果是，就會自動開始識別並且點擊。具體的點擊部分，筆者採用了作者為"wp"的一份代碼（從互聯網蒐集而得），裡面實現了基於win32api的窗體消息發送工作，進而完成了鼠標移動和點擊的操作。具體實現封裝在mouseOperation.py中，有興趣可以在文末的Github Repo中查看。

	时间	Date
初级	0.28	20:17:09 16.04.2018
中级	0.74	20:18:51 16.04.2018
高级	5.00	20:14:28 16.04.2018
Total	6.020	
	3BV/s	Date
初级	42.857	20:17:09 16.04.2018
中级	90.805	20:23:24 16.04.2018
高级	33.200	20:14:28 16.04.2018
Total	166.862	

- **項目完整代碼/GitHub地址| ArtrixTech/BoomMine**
- **作者/本文作者| Artrix**

下載1：OpenCV-Contrib擴展模塊中文版教程

在「

下載2：Python視覺實戰項目52講

在

下載3：OpenCV實戰項目20講

在

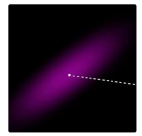
交流群

歡迎加入公眾號讀者群一起和同行交流，目前有 請按照格式備註，否則不予通過 添加成功後會根據研究方向邀請進入相關微信群。請勿

閱讀原文

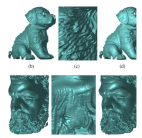
喜歡此內容的人還喜歡

經典重溫：卡爾曼濾波器介紹與理論分析



我愛計算機視覺

從零搭建一套結構光3D重建系統[理論+源碼+實踐]



3D視覺工坊

一篇文章總結出Opencv的知識體系

機器視覺課堂