驚喜的局部閾值二值化算法實現

新機器視覺 2022-03-06 20:30

以下文章來源於瘋狂的FPGA ，作者CrazyBingo

圖像二值化的目的

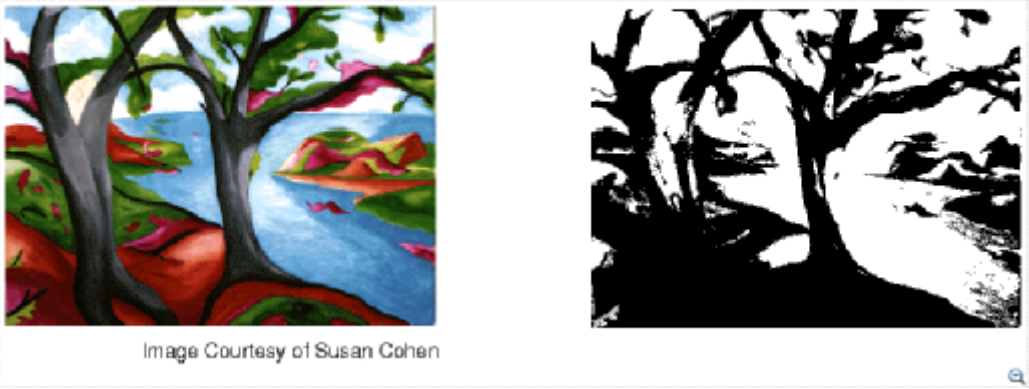
首先，視頻圖像的採集與處理，最終無非是兩個目的，或是供人查看審閱（視頻錄像、監控視覺等），或是供機器檢測識別（二維碼、車道線、物體姿態等檢測）。

對於人眼視覺，我們更關注圖像的色彩、清晰度、分辨力等，更多的我們希望在畫質上能夠給人帶來更加賞心悅目的感覺，所以手機很重要的一個指標就是拍照的效果——ISP，各家也都在ISP設計/ Tuning上花精力，來提高產品的競爭力。

但若是機器視覺，很多時候我們並不關心圖像本身的色彩甚至灰度圖像，而是具體的邊緣輪廓等，比如二維碼、車道線、運動檢測、區域分割等，再比如循跡小車更關注的是路線的識別與跟踪，因此通常我們只需要二值化圖像，最後再進行進一步的檢測。

Examples

```
load trees
BW = im2bw(X,map,0.4);
imshow(X,map), figure, imshow(BW)
```



以上圖為例，右圖為左圖的二值化結果，儘管存儲容量降低到了1/24，但是仍然不耽誤我們看整體形態，以及對數目的識別。

圖像的二值化，是後續圖像處理的基礎，我們從彩色或者灰度圖像上，獲得二值化圖像，然後在後續的圖像算法中進行進一步的識別處理。圖像二值化提取的質量，很大程度上決定了後續算法的效果與性能。

那麼，本章的核心，就是灰度圖像的二值化處理算法的實現。這裡由於篇幅的有限，前面全局閾值二值化的內容就不放出來了，我們直接進入進一步優化的主題：局部閾值二值化算法。

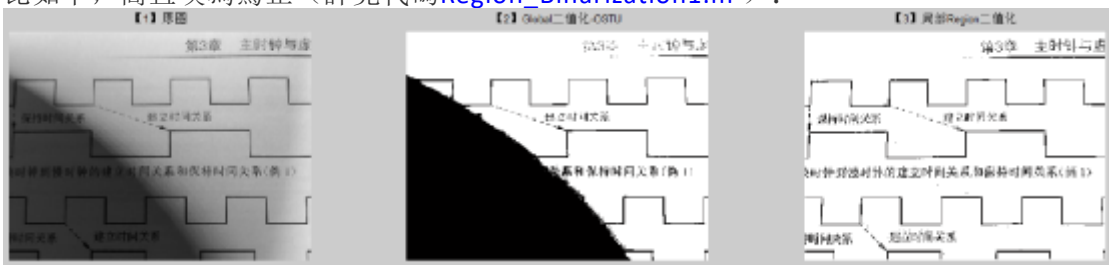
局部閾值二值化算法理論

算法本身其實沒有好壞，每一種算法都有其擅長的場景，相應的也有一些缺陷。

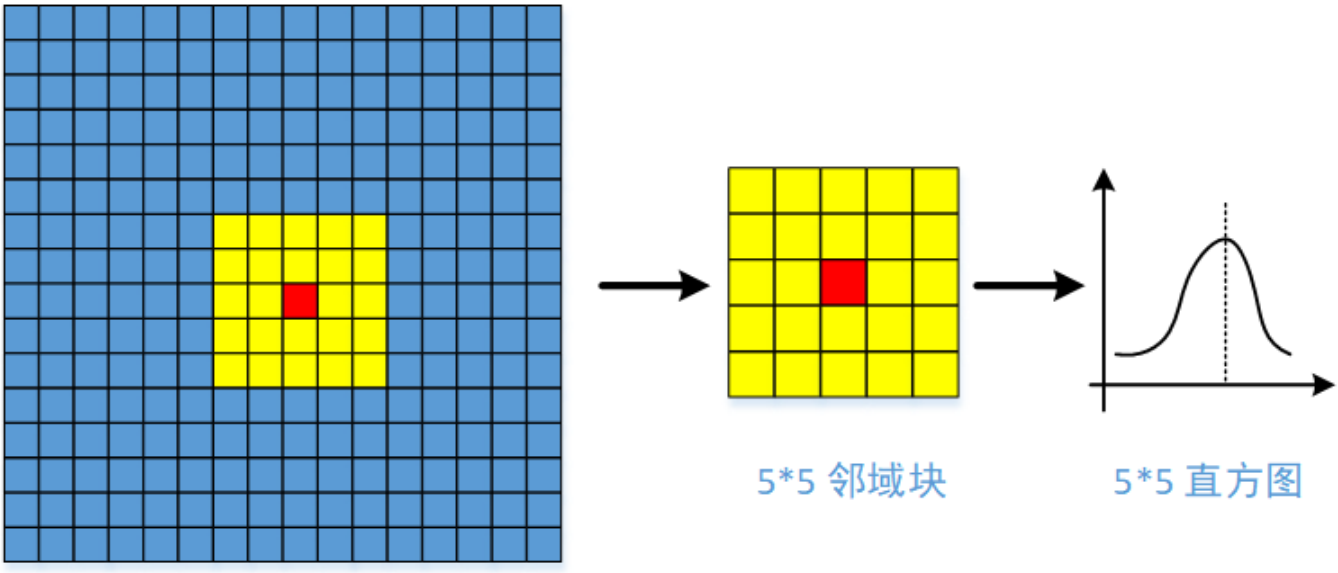
前面說的全局閾值二值化，對明暗均勻性比較好圖像有一定的優勢，並且計算量也很小。然而大千世界多姿多彩，圖像場景明暗不一，我們採用全局閾值二值化針對所有場景，其效果不一定理想。

進一步分析，圖像像素在空域上是一個二維的分佈，當前像素和其周邊像素具有高斯特性的相似度，而離的相對較遠的像素，其相干性就很小，甚至幾乎可以忽略不計。但在進行全局閾值計算的時候，所有像素的權重都是相同的，這必然將引入很大的誤差。採用以當前像素為中心的領域塊去計算局部閾值，其結果的可信度往往比全局閾值二值化，要高的多。

為此我隨手拍一個明暗分明的圖文場景，採用OSTU閾值二值化，以及採用5*5的局部閾值二值化，其結果對比如下，簡直嘆為驚止（詳見代碼Region_Binarization1.m ）：



其中圖一為真實場景中明暗分佈的圖片，採用OSTU全局閾值，其暗部的內容幾乎全軍覆沒，而採用局部閾值的方式，其結果並不受到原始圖像明暗的影響，相對較好的提取了圖中的字符紋理。這也將是本節要介紹的重點：局部閾值二值化算法。



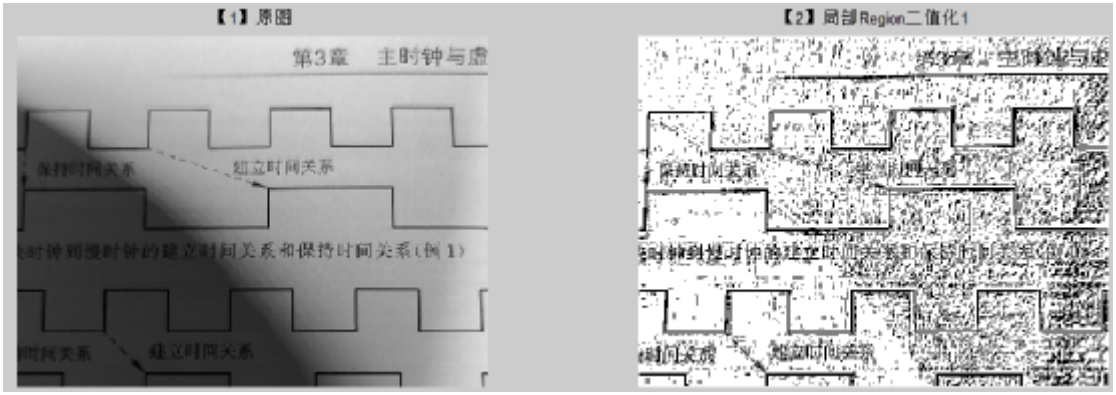
Matlab代碼的實現

局部閾值二值化的核心思想，大致如上圖所示。為了對紅色像素進行二值化，我們需要知道紅色像素鄰域的閾值。這裡假設採用5*5的鄰域塊，計算窗口內的閾值，然後根據閾值對紅色像素進行二值化。

這裡，如何計算窗口內像素的閾值，便成了局部閾值二值化的核心。類似全局閾值的計算，有以下幾種方法：

1）採用中值128；
2）採用OSTU計算窗口內像素的閾值；
3）計算窗口內像素的均值

這裡第一種又成了全局閾值二值化了，效果不理想；而第二種採用OSTU計算局部閾值，再遍歷全圖，這計算量比全局閾值還要大25倍，硬件不具備可實現性，軟件也無比耗時。那麼我們測試一下窗口內均值計算的方法，如下圖所示，慘不忍睹：



從上圖分析，確實我們在暗處也把紋理給提取出來了，但是同時也引入了很多的髒點。明明原本空白區域，多出了很多莫名其妙的異常點。我們假定5*5窗口內，沒有文字圖案，只有白色的紙張背景，那麼25個像素必然在其均值附近分佈，所以直接採用均值進行二值化，必然會有一些低於閾值的黑點產生。但這實際相對有效的紋理而言，這只是略微波動的背景噪聲而已，不該被當作有紋理圖案提取。

要解決這個問題，我們需要一個容錯的機制，或者說把這個條件稍微放寬一點，比如低於這個計算的閾值的一定比例，才算是有效的紋理。那麼我們引入一個對閾值進行縮放的參數（這裡主要是縮小），結合前面5*5區域計算均值進行局部閾值二值化，相關Matlab代碼如下所示（詳見region_bin_auto.m ）：

```
% 灰度圖像佈局自動二值化實現
% IMG為輸入的灰度圖像
```

```
% n為求閾值的窗口大小，為奇數
% p為閾值的縮放
function Q=region_bin_auto(IMG,n,p)


[h,w] = size(IMG);
Q = zeros(h,w);
win = zeros(n,n);

bar = waitbar(0,'Speed of auto region binarization process...');  %創建進度條
for i=1 : h
   for  j=1:w
      if(i<(n-1)/2+1  || i>h-(n-1)/2 || j<(n-1)/2+1 || j>w-(n-1)/2)
         Q(i,j)  = 255;    %邊緣像素不計算，直接賦255
      else
         win  =  IMG(i-(n-1)/2:i+(n-1)/2,  j-(n-1)/2:j+(n-1)/2);    %n*n窗口的矩陣
         thresh  = floor( mean(mean(win))) * p;
         if(win((n-1)/2,(n-1)/2)  < thresh)
            Q(i,j)  = 0;
         else
            Q(i,j)  = 255;
         end
      end
   end
   waitbar(i/h);
end
close(bar);   % Close waitbar.
```

核心代碼主要是n*n窗口內的均值計算，我們累加後計算均值，再乘上了一個縮放的參數，得到最後n*n窗口內的閾值，再通過比較得到二值化後的圖像。此時我們對比縮放前局部閾值二值化的結果，如下圖所示，髒點已經被去除，但文字圖案被很好的提取了出來：

見證了局部閾值二值化的優勢，我們再來扒扒局部閾值二值化的劣勢。上圖中，我們在5*5的窗口內，進行局部閾值的計算。但如果是更大或者更小的字符，5*5窗口是否還可以得到較好的結果，請看下圖：

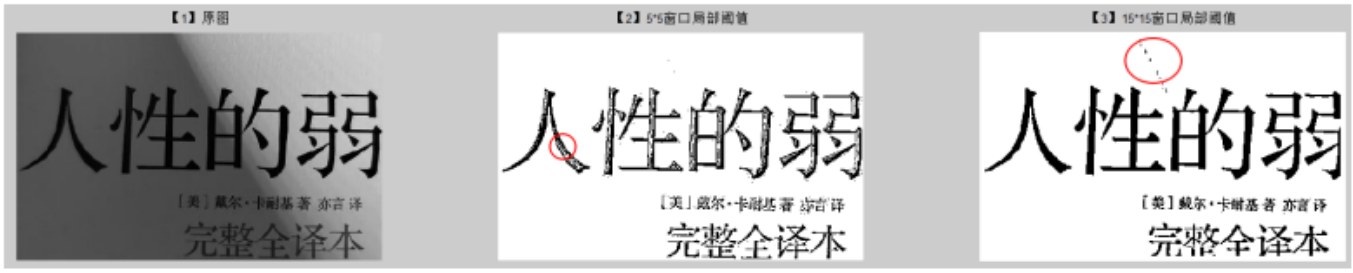


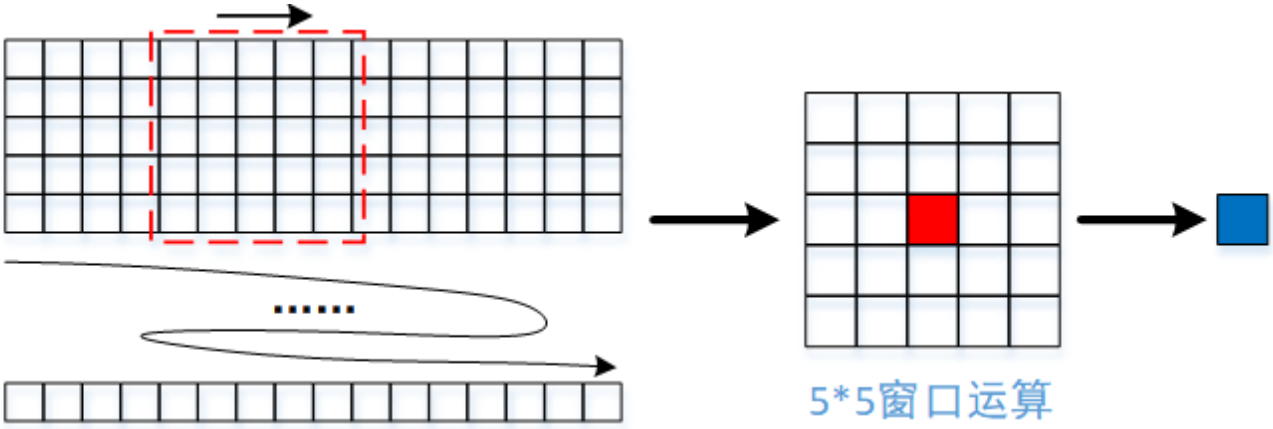
图 1　　图 2　　图 3

上圖中，圖2採用5*5的窗口進行局部閾值二值化計算，字符中間有被鏤空，同時也出現了零星的髒點，這主要是窗口太小，覆蓋在邊緣與字體時，陷入了局部最優，無法計算得到合理的閾值。而圖3採用15*15的窗口進行局部閾值的計算，由於窗口足夠大，不會出現局部最優，很好的解決了圖2的bug。

1）但這也引入了一個問題，如圖3圈圈所示，在明暗相間的區域，採用較大的窗口，錯誤的將分界線當作了圖案紋理。並且15*15窗口，進行全圖遍歷閾值的計算，這個計算量也不小，所以，算法並沒有嚴格的對與錯，想要做到真正的局部閾值自適應，還是有一定的難度。

FPGA硬件加速實現

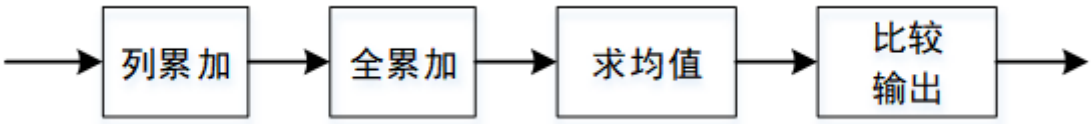
在前面高斯濾波中，我們採用行緩存，以流水線的方式實現5*5的濾波操作。這裡我們的局部閾值二值化算法，其實現流程和5*5的高斯濾波基本一致，仍然是一種通用的滑窗計算方法。



如上圖所示，在FPGA中採用行緩存，實現流水線5*5窗口的獲取，再採用流水線進行均值的計算，以及結合一定的縮放，得到最後的閾值，最後通過比較得到最終的二值化數據。這里相對5*5高斯濾波，只是多瞭如下幾步操作，其核心實現方法還是一樣的：

1）5*5窗口內的權重都為1，即均值計算
2）對均值進行縮放，上述Matlab代碼主要是縮小
3）對採用2）值作為閾值，比較得到結果，而非直接替代最後輸出

因此在FPGA開發中，我們可以基於前面5*5高斯濾波的RTL代碼，計算均值，並繼續流水線方式進行2）3）的操作即可。主要流水線流程如下所示：



這裡的均值涉及到了浮點操作，假設以縮放p=0.9為例，我們需要做一定的轉換，如下所示，我們巧妙的結合除法再乘以浮點的操作，通過放大及近似的方法，將其轉換為加法/移位操作，其中最後一步還可以直接去高8bit代替移位操作，節省LUT 。

$$\begin{aligned} thresh &= \frac{SUM}{25}\times 0.9 = \frac{SUM}{250}\times 9 \\ &\approx \frac{SUM}{256}\times 9 \\ &= (SUM\times 9) >> 8 \\ &= (SUM + [SUM, 3'b0]) >> 8 \end{aligned}$$

本文僅做學術分享，如有侵權，請聯繫刪文。

—THE END—

喜歡此內容的人還喜歡

CFD算法植入怪現象，球大佬指點

CFD界

聊天截圖厚碼也不安全，大神寫了算法分分鐘給你還原

量子位

馬賽克被破解了！大神研究出還原算法

Linux就該這麼學