二. 介紹

本次專題為手寫辨識四則運算實驗，在一張影像上用手寫出算式，使用影像處理技術將手寫部分偵測到，並且將其辨識為數字或運算符號，最後將辨識出來的結果連貫並且算出正確的答案。

三.怎麼做(逐步解釋):

1.

將訓練資料讀進來並且轉為灰階格式。

2.

因為當初訓練資料是用手機拍攝，且因為拍攝光源在影像上面，所以上面部分比較亮，下面部分比較暗，經過實驗發現這會影響最後的辨識結果，所以首先使用人工空間點的強化技術，將訓練資料圖片切成 8 列，並且將每列的背景部份由上到下依序加一個值，目的是使得每列的背景部分的值不會差太多，至於如何偵測是否為背景(白色部分)，因為手寫部分為黑色、背景為白色，兩者值肯定差很多，所以使用簡單的判斷式，if(image[i][j] > value1): image[i][j]+=value2，(image[i][j]為影像中某一像素、大於 value1 代表肯定為背景白色、value2 代表背景要加上得值)示意圖如下:

3.

將訓練資料影像使用高斯濾波進行平滑化(cv2.GaussianBlur(gray, (5, 5), 0)，kernal size 為 5x5，空間標準差為 0)。

將影像的白色背景再做二值化並且反轉(0~255) (cv2.threshold(blurred, 200, 255, cv2.THRESH_BINARY_INV)，像素點值>200 的改為 0，其餘變為 255)。

4.

將訓練資料切成 4 維(8x6x20x20)，每一筆訓練資料(x,y)，x:20x20 的像素 flatten 成一個 400 維的向量，y:為正確 label。

5.

此次訓練將

手寫'0':label 成數自字 0。

手寫'1':label 成數字 1。

手寫'2':label 成數字 2。

手寫'3':label 成數字 3。

手寫'+'號:label 成數字 10。

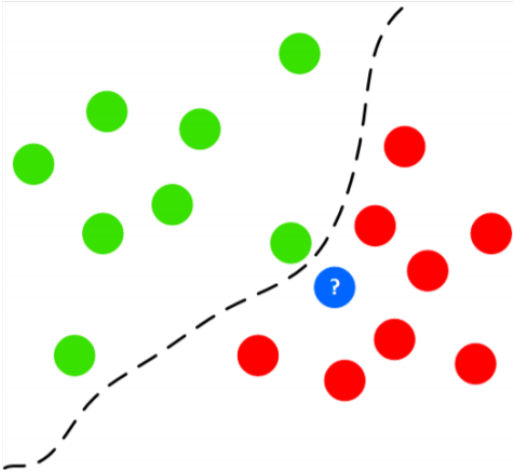
手寫'-'號:label 成數字 11。

手寫'x'號:label 成數字 12。

手寫'÷'號:label 成數字 13。

6.

使用 knn 分類演算法做訓練，knn 算法介紹如下:



假設我們現在要預測藍色點屬於哪一點，我們通常會選定一個奇數值 k，並計算出藍色點距離其他所有點的歐基里德距離，看看離藍色點最近的 k 筆資料是哪一類比較多，就直接將藍色點 assign 成那一類(k 為可調參數，本次實驗將 k 設為 1)。

以下為測試部分

7.

將測試資料讀出且用灰階表示，再使用高斯濾波(cv2.GaussianBlur(gray, (5, 5), 0)，kernal size 為 5x5，空間標準差為 0)(濾完波的結果會存下來，之後還要使用)，再將濾完波的結果丟進 canny operator 偵測邊(cv2.Canny())，而根據 opencv 官方文件顯示，此函數會做以下步驟:

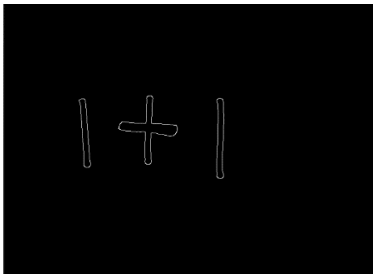
a. 先做 kernal size 為 5x5 的高斯濾波。

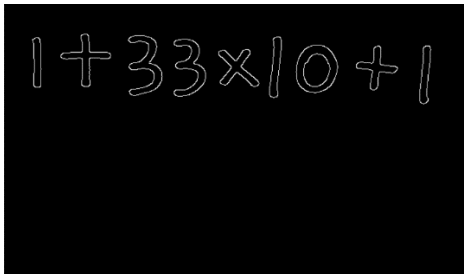
b. 再使用 Sobel kernel 計算每個像素的水平跟垂直的一次微分(Gx 跟 Gy)，所以一次微分的大小 Edge_Gradient(G) = sqrt(Gx*Gx+Gy*Gy)，方向為$\tan^{-1}(\frac{Gx}{Gy})$，而為了簡化計算，梯度的方向分成四種，將原本落在 0~22.5 度 or 157.5~180 度歸類在 0 度，22.5~67.5 度歸類在 45 度，67.5~112.5 度歸類在 90 度，112.5~157.5 歸類在 135 度。

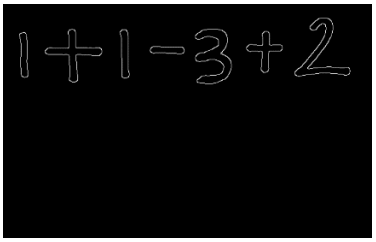
c. Non-maximum Suppression:因為上面 b 步驟所找出來的邊並不是只有一個像素的寬度，所以為了邊緣只有一個精確點的寬度(也就是只保留了梯度變化中最銳利的位置)，此步驟會依序去尋找每個點的梯度方向上變化最大的點，並且只保留 local 的最大值，例如:假設某像素的梯度方向為 45 度，那麼就觀察右上及左下的梯度大小是否大於此像素的梯度大小，若是則將此像素設為 0，若不是則將此像素的大小保留。

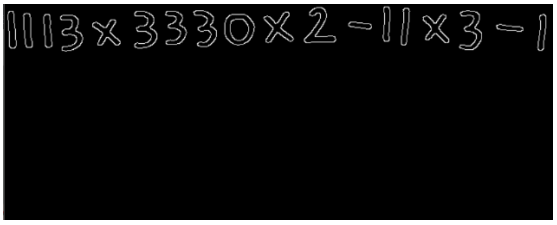
d. 自行設定兩個閥值(max,min)，依序尋找若像素點的梯度大小大於 max 則此像素就是最後輸出的邊點，若小於 min 則將此點捨棄不為邊點，若值介於中間的則看看此像素周圍 8 個點有沒有邊點，若有則也將此像素輸出為邊點，若沒有則將此像素點捨棄。(此步驟的邊界點輸出值都設為 255)

e. 最後輸出圖片及為結果。(此為測試資料的四張圖)









8.

使用 cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)去找尋輪廓(在 openCV 的世界裡， 若 Edge 線條

頭尾相連形成封閉的區塊，那麼它就是 Contour，否則就只是 Edge)，而這個函數的參數第一個 edged.copy()為剛剛 canny edge 偵測出來的結果，第二個參數 cv2.RETR_EXTERNAL 代表只偵測外圍輪廓，第三個參數 cv2.CHAIN_APPROX_SIMPLE 代表壓縮過後存取特定的一些輪廓資訊，例如:若輪廓的形狀為正方形那麼事實上不需要存取整個輪廓上的所有點，而是儲存四個頂點座標就夠了，接著為了後面的應用也就是辨識算式並計算答案，所以會先算出每個輪廓 x 方向的重心，並由左到右排序好在進入下一階段的疊代(為了從左至右依序辨識)。

9.

由找出來的 countours list 做疊代，每一個 countour 找出其 bounding box(也就是將一個輪廓用矩形框出來)(使用 cv2.boundingRect()函數)，若 bounding box 太小就會將其視為雜訊不考慮，cv2.boundingRect()函數的返回值資訊為 x 座標、y 座標、bounding box 的長、bounding box 的寬，根據這些資訊再回去抓已經濾完波的測試圖片，將位置抓到擷取出圖片，並且將擷取出來的部分 resize 成大小為 20x20 然後再將其轉為一個 400 維的向量，最後丟進之前訓練好的 knn 模型裡面去做分類，本次實驗在這裡將 k 值設為 1，最後輸出結果並用 list 儲存辨識結果。

10.

這一部分就剩下將 list 裡的辨識結果轉化為算式並計算最後答案，我寫的演算法如下:

假設現在有一算式:110×20+3×10，辨識完之後會 creat 一個 list[1,1,0,12,2,0,10,3,12,1,0](＋－×÷在此程式辨識為 10,11,12,13)，再另外 creat 兩個 list1[110,20,3,10](數字分離)、list2[12,10,12](符號分離)，先做成除法，A = list1[0]*list1[1]，算完之後把 list1[0]、list1[1]刪除，新增 A 到 list1[0] ，再把 list2[0]刪除(也就是刪除乘號)，依此類推算完成除法再算加減法，最後就可以 return 解答。

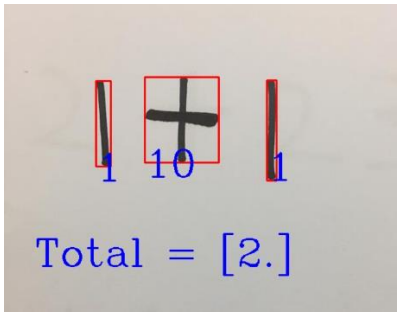
四.結果

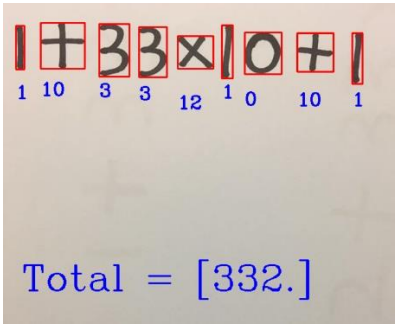
說明:

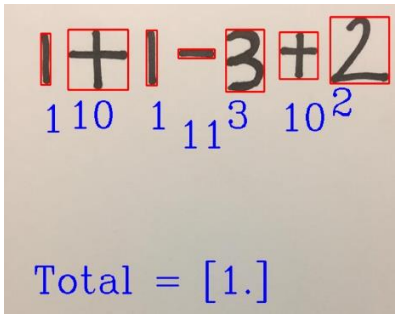
a. 紅色外框為 bounding box。

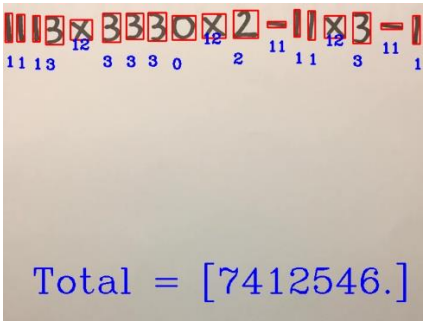
b. 每個紅色外框下面的藍色數字為辨識後的分類結果。

c. Total 為算式最後答案。









五.討論

本次實驗其實在我的測試之下發現其實用以上的演算法其實會出現一個問題，也就是其實這隻程式是無法辨識手寫 '÷' 的，因為我使用 cv2.findContours()這個函數把每個輪廓當作一個要去辨識的物件看待，所以當手寫出 '÷' 時，事實上會有三個輪廓('·', ' - ', '·')，這樣的結果並不是我所想要的，但是我臨時也想不出怎麼把整個'÷'當作是一個物件去辨識，所以還是使用了這個方法。

再來就是其實辨識率並不好我在想可能 knn 演算法過於簡單以及訓練資料量太少導致這種結果產生。

以上兩個部分就是未來可以再作改進的部分，讓這個應用更加完善，並且效率更好。