

# iPhone Sudoku Solver 제작

2018-07-16

남상규

[halite@gmail.com](mailto:halite@gmail.com)

## 목차

1. 배경.....	3
2. 어떻게?.....	3
3. 우선 해결 해야할 것 들.....	5
3.1. 수도쿠 영역 인식 방법 .....	5
3.1.1. 어떻게?.....	6
3.1.2. Python Code.....	15
3.2. 숫자 인식 MACHINE LEARNING .....	18
3.2.1. MNIST.....	18
3.2.2. 입력 데이터 .....	20
3.2.3. 학습.....	22
3.2.4. Model.....	23
3.2.5. 테스트.....	24
3.2.6. 그런데... ..	25
3.3. 수도쿠 풀이 알고리즘 .....	29
3.4. IPHONE에 MACHINE LEARNING 결과 돌리기 .....	39
3.4.1. CoreML Model 변환.....	40
3.4.2. XCode에서 사용.....	42
3.4.3. CoreML Model 사용.....	43
4. IPHONE 앱 제작.....	43
4.1. 동작 구상 .....	43
4.2. 카메라 동작 .....	44
4.2.1. 카메라 준비.....	45
4.2.2. 카메라 출력 처리.....	46
4.3. 아래 UIImageView 터치 이벤트 처리 .....	48
4.4. 수도쿠 풀이.....	49
4.4.1. 풀어진 수도쿠 화면에 표시.....	51
4.5. IPHONE에서 OPENCV 사용하기.....	51
4.5.1. 준비.....	51
4.5.2. OpenCV Wrapper 등록 .....	52
4.5.3. Wrapper 수정 .....	53
4.5.4. OpenCV 코드 작성.....	54

## 1. !

이 문서는 필자가 취미로 만든 앱과 거기에 들어간 꼼수 등에 대해 간략하게 설명한 것으로 취미 개발에 따른 의식의 흐름을 보여준달까? 뭐 큰 기대는 하지 마시고 그냥 가볍게 보는 정도로만 여겨 주었으면 함.

특히 코딩 기술이 좋은 편이 아니라 어떻게 하면 구현을 할 수 있을까? 뭘 기술을 조합하면 내가 원하는 것을 할 수 있을까?가 주 관심사이므로 코딩 기술에 대해서는 너무 딴지 걸지 마시길...

모든 코드는 <https://github.com/mrhalite/Sudoku-Solver.git> 에 있음.

## 2. 배경

아내가 수도쿠를 즐기다가 가끔 안 풀리면 나에게 요청해 풀어달라는 일이 있었다. 뭐 나라고 해서 수도쿠의 달인이거나 천재는 아니어서 그런지 어려운 문제는 풀어주지 못했고 때로는 그 것도 못 풀고 맨날 잘난 척만 한다는 핀잔을 들었다.

때마침 취미 거리로 Machine Learning에 대해 공부하고 있기도 해서 아내가 수도쿠를 풀어 달라고 하면 핸드폰으로 수도쿠를 찍어 풀이 해주는 앱을 만들면 좋겠단 생각을 했다. 만들면서 기대하는 것은 입력 이미지 신호처리, Machine Learning으로 문자 인식을 하는 것, Machine Learning으로 만들어진 Network을 iPhone에 적용하는 것, iPhone 프로그래밍 하는 방법을 익힐 수 있을 것이라 생각했다.

공부도 하고 내 귀차니즘도 풀고 1석2조랄까? (그 보다는 뭔가 만드는데 좋아서...)

## 3. 어떻게?

하겠다는 생각은 했으니 이제 어떻게 만들 것인지 구상을 해볼까? 거창하게 UML 등을 써서 그럴싸 하게 만들면 좋겠지만 취미 생활에 그렇게 까지 하는 것은 도끼로 모기 잡는 것이니 그렇게 하지는 말고(사실 UML을 모른다 T\_T)... 그래서 아래와 같은 요구 사항을 정리 해봤다.

- ① 대략적으로 생각한 것은 수도쿠를 풀어달라는 요청이 언제 어디서 있을지 모르니 항상 휴대 가능한 기기로 풀어줄 수 있어야 한다. → 항상 휴대 가능한 것은 핸드폰 밖에 없네? 그럼 핸드폰으로 디바이스는 잠정 결정!
- ② 핸드폰을 사용해 수도쿠 퍼즐을 풀려면 와이프가 풀고 있는 수도쿠 퍼즐을 입력 받아야 한다. → 핸드폰 카메라로 찍어 앱에 입력해 주면 되겠군!
- ③ 동작 방식은 이렇게 하면 될 것 같음 : 핸드폰 카메라로 수도쿠 촬영 → 영상 처리로 수도쿠 영역 인식 → 수도쿠 영역에서 이미 주어진 숫자 이미지 추출 → 숫자 인식 (Machine Learning) → 수도쿠 풀이 → 화면 표시
- ④ 수도쿠 영역 파악은 영상처리가 필요하니 그 전에 자주 쓰던 OpenCV를 사용해 영상 처리 진행 (영상 처리를 하지 않고 Machine Learning으로 수도쿠 영역을 인식하게 만들 수

있지 않나? 이 것도 해보자)

- ⑤ 숫자 인식은 MNIST를 기반으로 Train Data를 잘 제공해 Python Model을 train후 weight를 만들자! (사실 iPhone에 Python으로 만든 Network을 넣을 수 있다는 것을 들어 알고 있었기에 이렇게 진행함. 몰랐더라면 stackoverflow를 뒤졌겠지...)
- ⑥ 앱 UI는 나중에 생각하자. 나 혼자 쓰는 건데 UI 좋을 필요 없다. 그리고 아내가 쓰게 되면 직관적으로 쓸 수 있어야 하니 UI는 최소화 하고 최대한 자동화 한다. (이게 제일 힘든 일인데...)
- ⑦ 수도쿠는 종류가 많은데 일단 9x9 수도쿠 만을 대상으로 한다.

이정도 구상을 하고 앱을 만들기 전에 우선 해결 해야할 것들을 test program을 작성해 가능성을 살펴 본 후 확정되면 그 방식을 따르고 잘 안되거나 틀린 생각이면 빨리 다른 방법을 찾아 진행하는 식으로 접근 했다(틀리더라도 계획을 세우고 실행하면서 틀린 것을 재빨리 수정하는 식으로 자꾸 반복해 일을 처리하는 것을 좋아해 처음부터 모두 정리하거나 거창하게 계획을 세우지는 않는다)

우선 해결 해야할 것 들은 다음과 같이 정리 했다.

- ① 수도쿠 영역 인식 방법
- ② 숫자 인식 Machine Learning
- ③ 수도쿠 풀이 알고리즘
- ④ Python으로 짤다

시작 단계에서는 ②번의 숫자 인식 Deep Learning이 힘들 것이라 생각했는데 해보니 ①번의 수도쿠 영역 인식이 훨씬 훨씬 더 힘들었고 아직도 개선해야 할 것이 많다고 생각되고 오히려 숫자 인식은 Train Data만 잘 만들면 엄청 쉬웠다. 이런 이유 중 하나는 Machine Learning이라고 거창하게 이름을 붙이긴 했지만 최근의 사물 인식 Machine Learning 등에 비하면 아주 아주 간단한 Network이기 때문에 그런 것이다. (이걸 나중에 해보고 알았다)

그에 비해 수도쿠 영역 인식은 처음엔 OpenCV 만으로 처리 했는데 그 결과가 그리 만족스럽지 않아 2주정도 테스트, 고민 하다가 방향을 바꿔 수도쿠 영역 인식을 Machine Learning으로하기로 맘 먹었다. 그리고 실제로 수도쿠 영역 인식을 주워 들은 Machine Learning 기술을 조합해 시도 해봤는데 그리 실통치 않았다(사실 완벽하게 동작 하리라 생각했는데 전혀 그렇지 않았다). 온갖 방법을 동원해 한달 정도 시도하다가 이걸 쉽지 않거나 뭔가 다른 대안이 없으면 현 단계에서 사용할 방법은 아니란 결론을 얻은 후(이 방법으로 너무 해보고 싶어 붙들고 있기는 했음) OpenCV 만으로 수도쿠 영역을 알아내기로 하고 이전의 시도에 약간을 더 넣어 한 이틀 만인가 만에 굉장히 잘 작동하는 놈을 만들어 냈다(하~ 괜히 돌아 왔네... 저 산은 아닌가벼!).

Python으로 짜기로 한 것은 이래저래 고려 할 것이 적기도 하고 내가 아는 빨리 테스트 해보는 방법 중 가장 좋은 방법이라 테스트는 Python으로 만들기도 했다. Python이 웬만한 기능을 다 제공하기 때문에 쉽게 접근이 가능하고 UI 같은걸 만들 필요 없어 기능 본연에 충실할 수 있기 때

문제 개인적으로 선호 한다.

## 4. 우선 해결 해야할 것 들

### 4.1. 수도쿠 영역 인식 방법

[www.dailysudoku.com](http://www.dailysudoku.com)에서 매일 올라오는 수도쿠 퍼즐을 보면 다음과 같은 화면을 볼 수 있다.

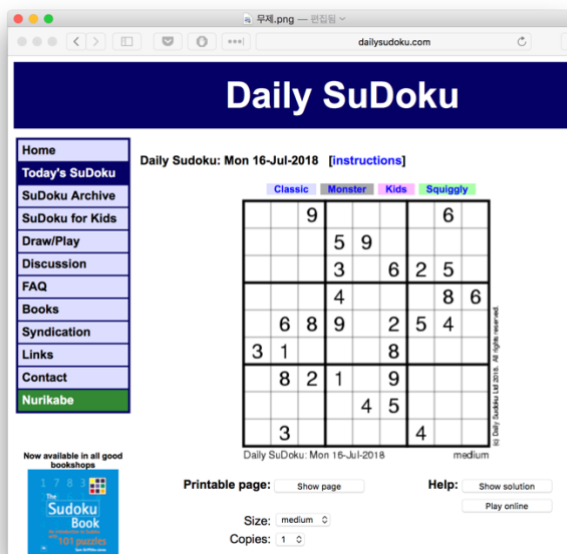


그림 1. 수도쿠 사이트 이미지

		9					6	
			5	9				
			3		6	2	5	
			4				8	6
	6	8	9		2	5	4	
3	1				8			
8	2	1		9				
				4	5			
3						4		

그림 2. 수도쿠 영역

그림 1을 핸드폰 카메라로 찍으면 그림 2와 같은 수도쿠 영역만 찾아내는 것이 가장 먼저 해야 할 일이다. 혹은 수도쿠 퍼즐 책을 핸드폰 카메라로 찍는 경우도 마찬가지로 수도쿠 영역만 찾아내는 방법이 필요하다.

이렇게 찾아진 수도쿠 영역을 이미지로 얻고 9x9 등분해 이미지를 쪼개면 81개의 작은 이미지를 얻는데 이 작은 잘라낸 이미지를 숫자 인식 Machine Learning에 넣어 숫자를 인식하도록 하겠다는 것이다.

원래 계획은 최 외곽의 가장 큰 수도쿠 영역을 찾고 그 안의 각각의 작은 사각형을 찾아 찾아진 영역에서 숫자 이미지 만을 추출 하려했는데 생각보다 쉽지 않았고 안정적으로 동작하게 만들기 쉽지 않아 전체 영역만 추출해 내고 그냥 9등분 해버리는 식으로 처리 했다. (대신 숫자 인식하는 단계에서 여러 가지 경우를 고려하면 될 것으로 추측했다. 나중에 알게 됐지만 숫자 인식 단에서도 한계가 있어 되도록 앞에서 입력되는 이미지를 최대한 좋게 만들어 주는 것이 인식률을 높이는 것임을 알게 됐다)

#### 4.1.1. 어떻게?

카메라로 찍은 이미지를 사용하기 전에 우선 수도쿠 사이트에서 이미지를 캡처해 그 이미지를 가지고 방법을 찾는 것이 먼저고 찾아진 방법을 카메라 출력 이미지에 적용해 튜닝 혹은 수정하는 식으로 계획을 세웠다.

입력 이미지에서 특정 패턴을 추출하는 방법은 여러 가지가 있으나 내가 이전의 경험으로 알고 있는 방법은 OpenCV를 사용해 contour를 찾아내고 contour에서 최외곽의 네 꼭지점을 선택하면 대략적인 수도쿠 영역을 얻을 수 있다는 것이었다. 혹은 contour를 얻고 직선을 얻어 직선들의 교차점을 찾아 최종적으로 최외곽 사각형 영역을 알아낼 수도 있지만 복잡하고 controur 최외곽을 찾는 것과 비교해 실험상 이득이 없어 사용하지 않았다.

OpenCV를 사용해 최외곽 꼭지점을 찾는 순서는 다음과 같다.

- ① 입력 이미지를 gray scale로 변환
- ② blur 혹은 histogram equalization (둘 다 필요 없을 수도 있으므로 실험으로 확인 필요)
- ③ threshold(binary invert) 적용
- ④ contour 찾기
- ⑤ contour 중 가장 큰 contour 찾기
- ⑥ 가장 큰 contour의 가장 왼쪽, 오른쪽 x 좌표 값 얻기. 그리고 가장 위, 아래 y 좌표 값 얻기.
- ⑦ 4개 값으로 최외곽 꼭지점 4개 만들기
- ⑧ 입력 이미지에서 4개 꼭지점으로 만들어지는 이미지 추출(crop)

웹페이지에서 얻어지는 이미지 등은 카메라로 찍지 않아 왜곡되지 않아 ⑧까지만 하면 되지만 나중에 카메라를 사용해 찍은 이미지에서 수도쿠 영역을 찾아내려면 다음과 같은 작업을 추가로 해야 한다.

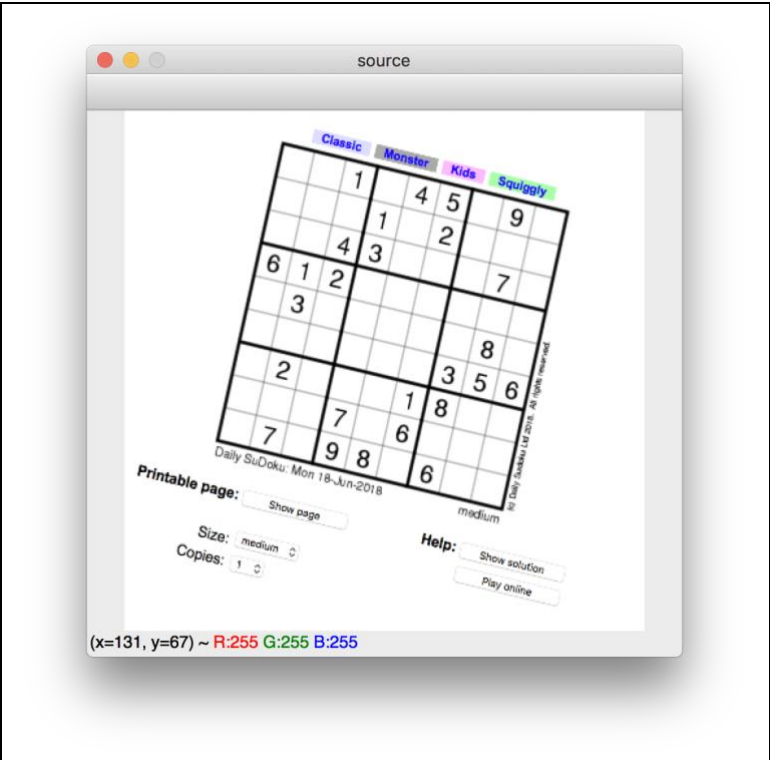
- ⑧ 4개 꼭지점으로 이루어진 찌그러진 사각형을 256x256과 같은 정사각형으로 변환 해주는 matrix 계산(OpenCV의 transform 기능)
- ⑨ 얻어진 변환 matrix로 4개 꼭지점으로 둘러싸인 이미지를 정사각형으로 변환

사실 최 외곽의 4개 꼭지점만 찾아내면 나머지는 쉬운 일이고 여기까지 얻어 내는 데에 시행착오나 실제 환경에서는 튜닝을 많이 해야 한다.

우선 위의 순서에 따른 이미지가 어떻게 되는지 확인해 보자.

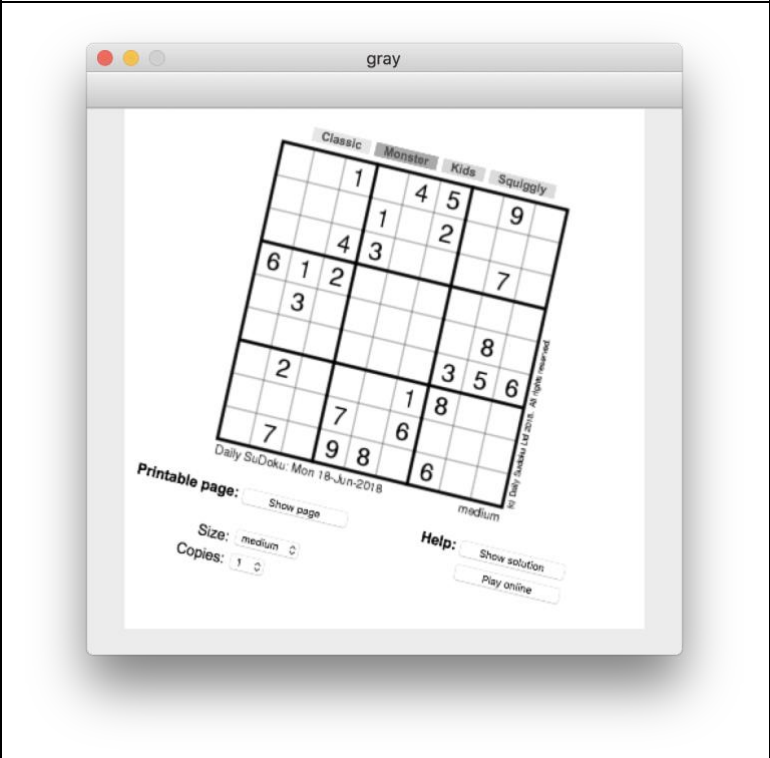
순서	이미지	설명
----	-----	----

1



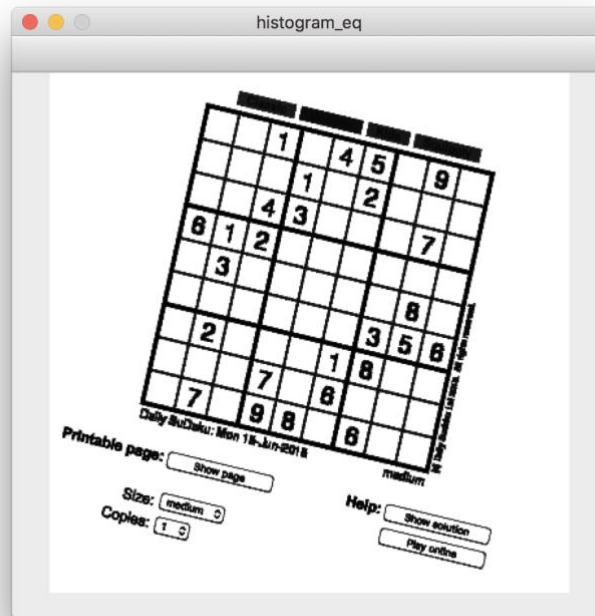
입력 받은 그림

②



입력	이미지를	gray
scale로	변환	

③

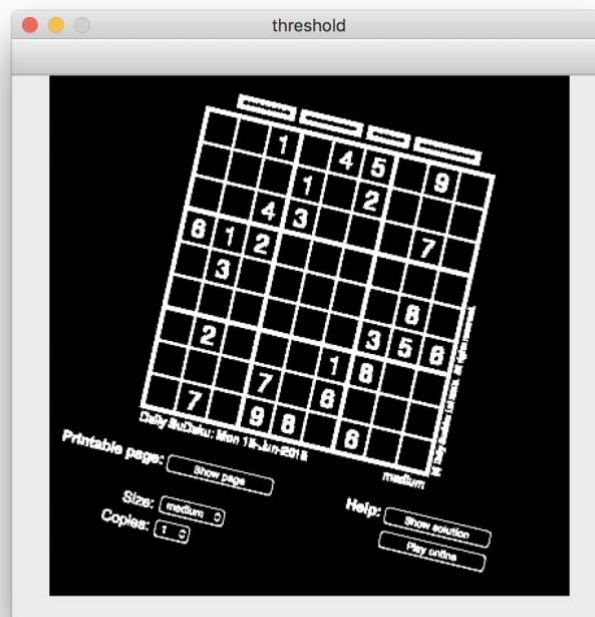


Histogram Equalization  
실행.

입력 이미지가 카메라로  
찍은 것이라면 contrast  
를 정리 해줘 좀 더 보  
기 좋은 이미지로 만들  
어 준다.

(실험적으로 필요한 단계  
인지 확인 필요하다. 보  
통은 필요 없다)

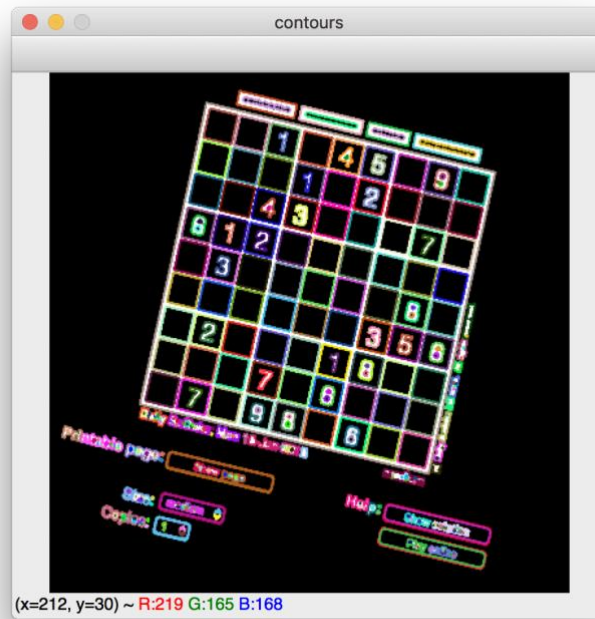
④



Binary Invert Threshold  
를 적용해 흰바탕에 검  
은색인 글자나 그림의  
실제 인식해야 하는 것  
들을 흰색으로 바꿔준다.  
Threshold Level을 조절  
해 노이즈나 원하지 않  
는 부분은 모두 0(검은  
색)이 되도록 하는 것이  
중요하다(테스트 이미지  
는 깨끗해 threshold가  
그냥 반전된 것으로만  
보이지만 카메라 이미지  
등에서는 양상이 다르  
다). Threshold Level은 실험적으로 잘 선택해야  
한다.

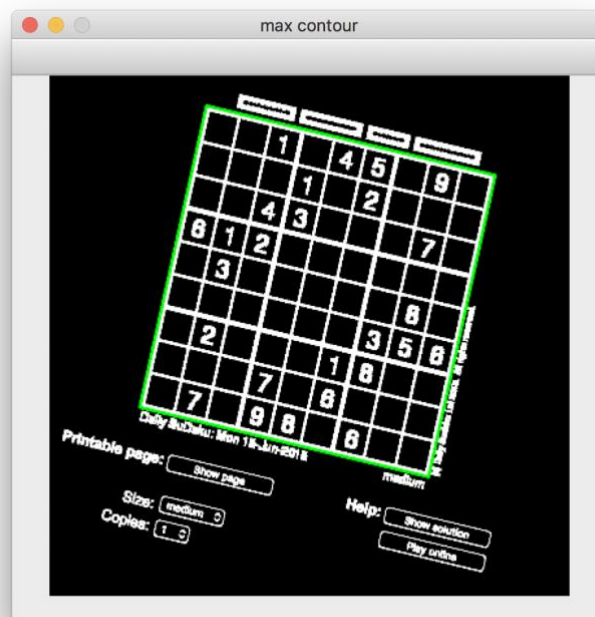


⑤



contour를 찾아 표시한 이미지(작은 것에서 큰 것 까지 색깔을 달리하면서 표시한 것)

⑥



contour 중에 제일 큰 것을 찾아 표시한 이미지

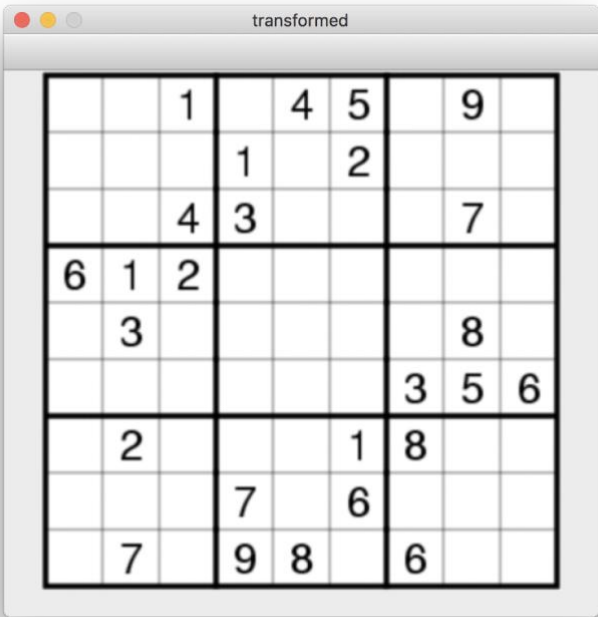
⑦		<p>최대 contour 영역을 정사각형 이미지로 변환한 것. 이 이미지가 최종적으로 찾아낸 수도쿠 영역이다.</p>
---	---	---

표 1. 수도쿠 영역 찾기 - 각 단계별 이미지

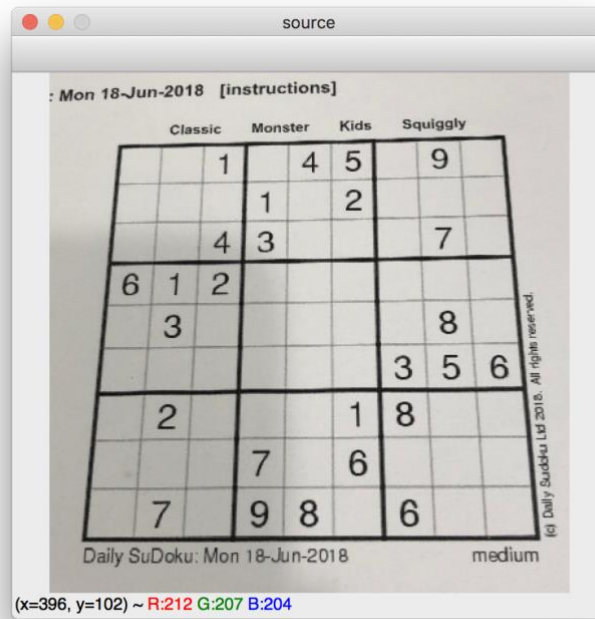
2번째 단계인 blur 혹은 histogram equalization은 필요한지 실험적으로 결정해야 한다. 보통의 경우 blur는 필요하고 histogram equalization은 필요 없는데 blur를 하면 흑시나 붙어 있어야 하는 부분이 카메라 촬영 등으로 떨어지거나 플래시 혹은 형광등 불 빛 등으로 밝게 비춰 끊어지는 것들을 조금은 연결해줘 threshold 후의 결과가 좋게 만들어 준다. 그러나 iPhone 카메라를 사용해 최종적으로 만들어진 앱에는 blur와 histogram equalization 모두 들어가지 않았다.

각 단계별 OpenCV 기술은 [www.opencv.org](http://www.opencv.org)나 [www.stackoverflow.com](http://www.stackoverflow.com)에서 찾아 내용을 숙지 하기 바란다.

아래는 같은 단계를 웹캠으로 받은 이미지에서 어떻게 동작하는지 테스트 해본 것이다(iPhone 카메라를 사용하기 전 웹캠에서 어떻게 동작하는지 미리 확인해 본것).

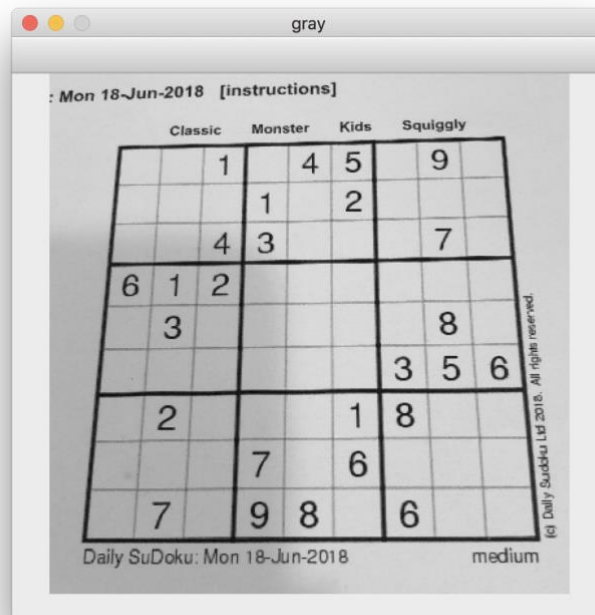
순서	이미지	설명
----	-----	----

①



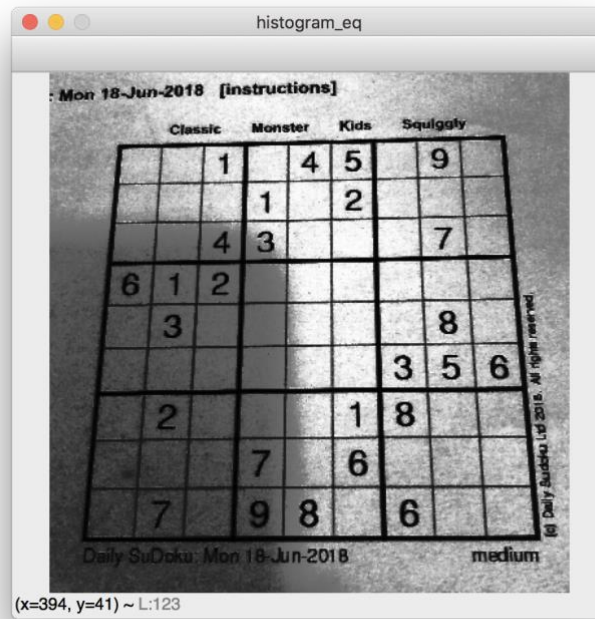
웹캠으로 입력 받은 그림

②



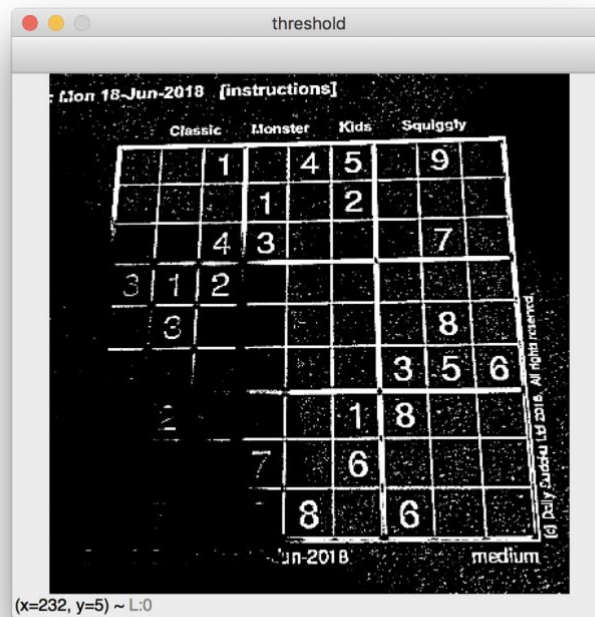
입력 이미지를 gray scale로 변환.  
입력 이미지에 이미 그림자 영역이 있어 이런 이미지에 대해 이후 단계에서 처리를 잘 해줘야 한다.

③



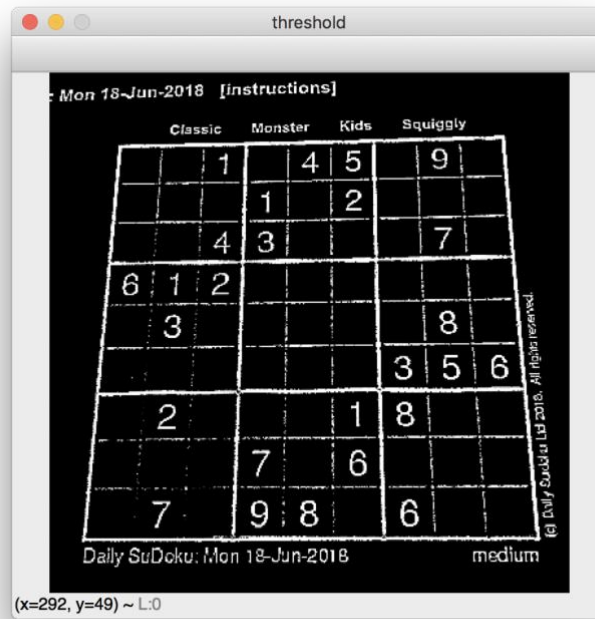
Histogram Equalization 실행. 이 경우는 오히려 역효과를 불러 일으켰다. 그러므로 웹캠에서 받은 이미지의 경우 이 단계를 빼는 것이 나을 것으로 보인다.

④



Histogram Equalization 된 이미지를 Threshold 취하면 이렇게 필요한 정보를 날려먹어 이전 쓸모 없는 결과를 낸다.

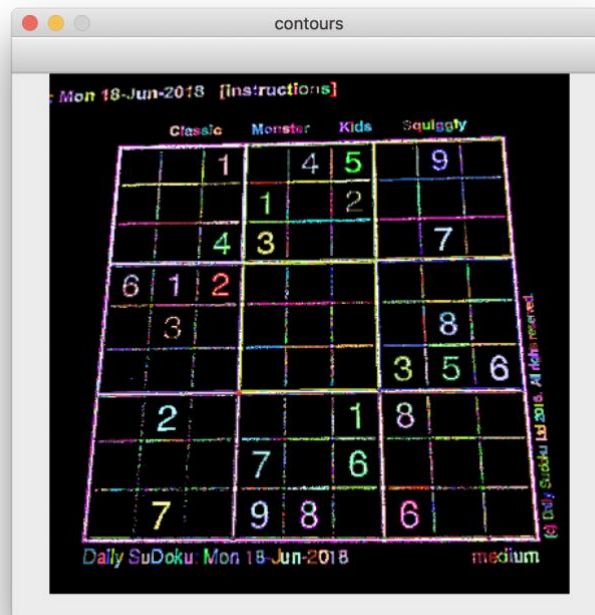
⑤



Histogram Euqalization  
을 하지 않고 Threshold  
를 한 이미지.

최 외곽을 찾기에는 충  
분해 보인다.

⑥



contour를 찾음

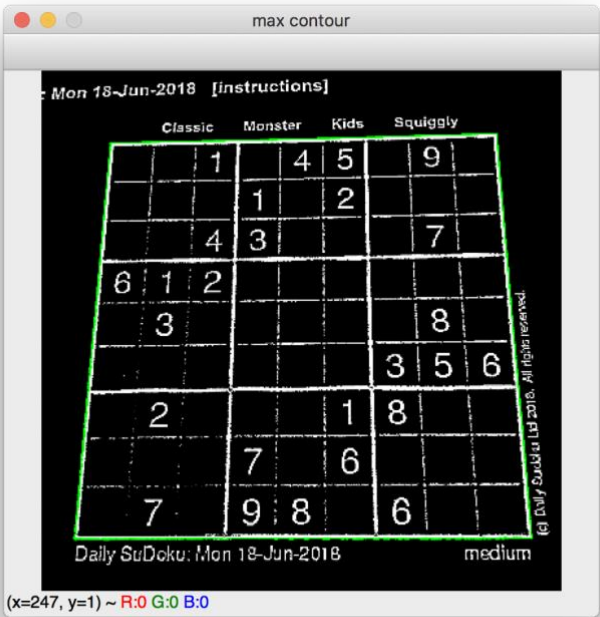
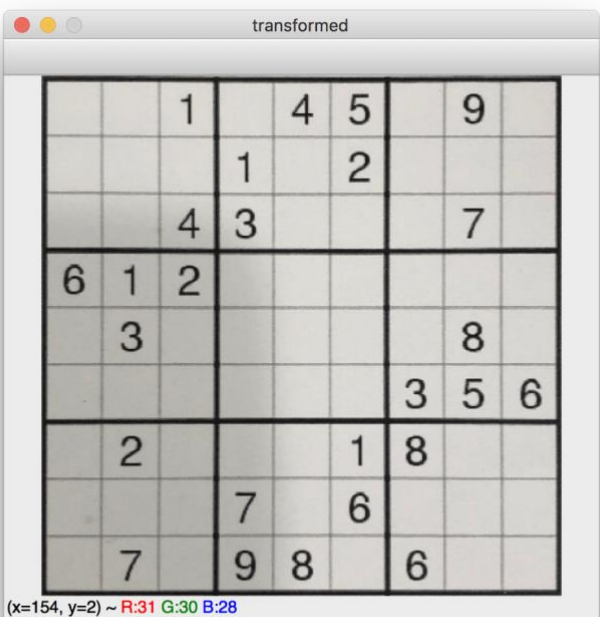
⑦		<p>최 외곽 contour를 찾았다.</p>
⑧		<p>최 외곽 영역을 정사각형 이미지로 변환한 이미지. 꽤 잘 찾아지는 것으로 보서는 blur 및 histogram equalization이 없는 것이 맞다.</p>

표 2. 웹캠 입력 - 각 단계별 이미지

대략적으로 수독구 영역을 찾아내는 방법은 정리가 됐고 실제 적용 단계에서의 튜닝(Threshold Level 값 선택 등)은 미래의 일로 남겨 두었다.

#### 4.1.2. Python Code

```
def getSudokuArea(srcMat):  
    """ get sudoku rect """  
    # convert to gray  
    grayMat = cv2.cvtColor(srcMat, cv2.COLOR_RGB2GRAY)  
  
    # histogram equalize  
    eqHistoMat = cv2.equalizeHist(grayMat)  
  
    # threshold  
    threshMat = cv2.adaptiveThreshold(grayMat, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY_INV, 31, 31)  
  
    # find contours  
    contourMat, contours, hierarchy = cv2.findContours(  
        threshMat,  
        cv2.RETR_CCOMP,  
        cv2.CHAIN_APPROX_SIMPLE  
    )  
  
    # find max contour  
    maxArea = 0  
    maxContourIndex = 0  
    for i, contour in enumerate(contours):  
        area = cv2.contourArea(contour)  
        if area > maxArea:  
            maxArea = area  
            maxContourIndex = i  
    maxContour = np.squeeze(contours[maxContourIndex])  
  
    # find rect points from max contour  
    rect = np.zeros((4, 2), dtype=np.float32)  
    s = np.sum(maxContour, axis=1)  
    rect[0] = maxContour[np.argmin(s)] # top left  
    rect[2] = maxContour[np.argmax(s)] # bottom right  
    d = np.diff(maxContour, axis=1)  
    rect[1] = maxContour[np.argmin(d)] # top right  
    rect[3] = maxContour[np.argmax(d)] # bottom left  
  
    """ crop sudoku area """  
    # get width, height of transformed image  
    (tl, tr, br, bl) = rect  
    widthA = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2))
```

```

widthB = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2))
heightA = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2))
heightB = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2))
maxWidth = max(int(widthA), int(widthB))
maxHeight = max(int(heightA), int(heightB))

# get transformation matrix & transformed image
dst = np.array([
    [0, 0],
    [maxWidth - 1, 0],
    [maxWidth - 1, maxHeight - 1],
    [0, maxHeight - 1]], dtype = "float32")
M = cv2.getPerspectiveTransform(rect, dst)
warpMat = cv2.warpPerspective(srcMat, M, (maxWidth, maxHeight))

return warpMat

```

코드 1. 수도쿠 영역 찾기 - 주요 코드

코드 1에서 Histogram Equalization은 필요 없으니 빼면 된다.

Threshold 단계의 maxVal, blockSize, C 값은 튜닝이 필요한 값이다. maxVal의 경우 255로 하면 되고 실제 중요한 것은 blockSize와 C 값인데 blockSize는 adaptive threshold 함수의 방식이 ADAPTIVE\_THRESH\_MEAN\_C이므로 MEAN을 취하는 크기라 보면 되고 크기가 클수록 두루뭉실하게 MEAN을 취해 끊어짐이 덜하다. 그리고 C 값은 MEAN을 취한 값에서 빼주는 값인데 일정 값 이하의 값은 모두 0이 되게 해준다. 즉 이미지에 노이즈가 있다면 C 값을 조절해 노이즈를 제거할 수 있다. 그림 3은 threshold에서 C를 31로 설정한 것이고 그림 4는 C를 2로 설정한 것이다. 보는 것과 같이 C 값이 크면 쓸데 없는 노이즈를 제거 해준다. 그러나 C 값이 커짐에 따라 필요한 영역의 신호도 같이 제거 되므로 계속 키울 수록 수도쿠 사각형의 연결 부위가 떨어지게 되어 사각형 검출이 안되는 문제가 발생한다. 그러므로 실험적으로 어느 정도의 값을 설정할 필요가 있다.



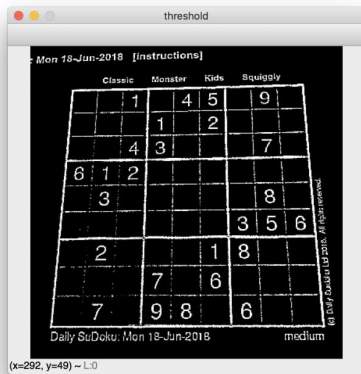


그림 3. C=31

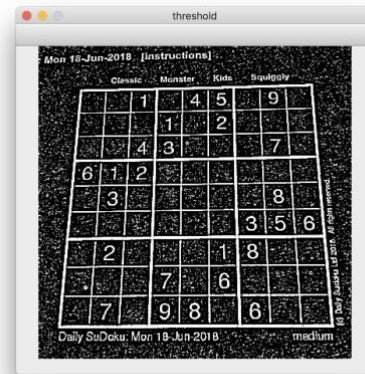


그림 4. C=2

contour를 찾은 후 제일 큰 contour를 찾는 방법은 각 contour의 면적을 비교해 가장 면적이 큰 것을 고른다. 수도쿠 영역에서 제일 큰 contour는 제일 바깥 사각형이 될 것이다.

최 외곽의 contour를 그린 그림이 그림 5인데 사각형을 찾은 것 같으나 실제로는 polygon을 찾은 것으로 필요한 네 귀퉁이의 점을 찾아 줘야 한다.

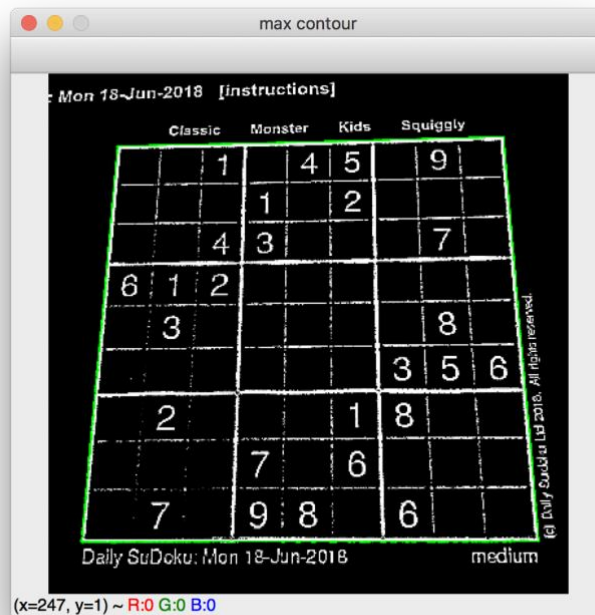


그림 5. Max Contour

max contour는 point의 array이다. 각 point의 x, y 값을 서로 더 하고 그 중에 가장 작은 값이 Top Left 점이 된다. 그리고 가장 큰 값이 Bottom Right 점이 된다. 왜 그런지는 각자 생각해 보길 바라고... 이번에는 x - y 값에서 가장 작은 값이 Top Right가 되고 가장 큰 값이 Bottom Left가 된다.

이렇게 얻어진 최 외곽 4꼭지점을 사용해 이 영역이 모두 포함되는 제일 작은 크기의 정사각형 크기가 얼마인지 알아낸다. 찌그러져 있거나 틀어져 있는 사각형을 정사각형으로 변환하는데 변환에 따른 오류를 최소화 하려면 변화량이 최소화 되도록 해야하는데 생각한 방법이 그림 6과 같이 찾은 4 귀퉁이를 사용해 틀어져 있는 사각형이 들어가는 최소 크기의 사각형을 만들고 거기에 맞게 변환 한다.

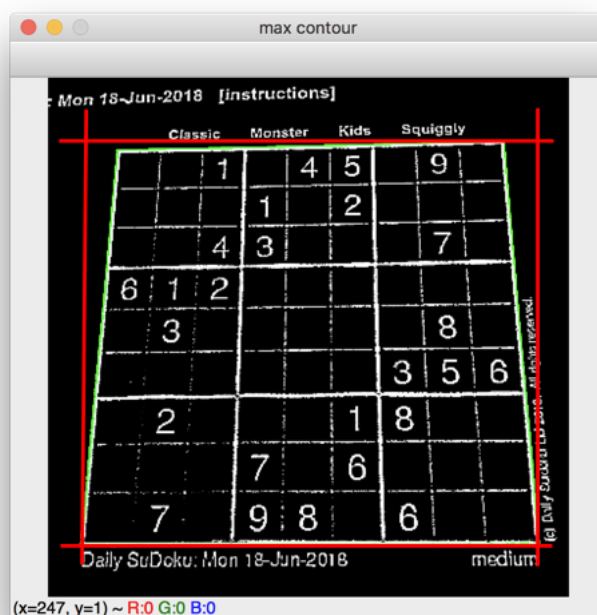


그림 6. 변환을 위한 최소 크기 사각형

Python code에서 warpMat가 얻어진 최종 수도쿠 영역 이미지이다.

#### 4.2. 숫자 인식 Machine Learning

숫자 인식을 Machine Learning으로 해결하기로 했기 때문에 관련된 내용 들을 찾아 보았는데 기본적으로 Machine Learning이 입력 데이터를 Network을 통과 시켜 원하는 결과를 얻는 방식으로 되어 있기 때문에 입력 데이터와 Network을 어떻게 구성하면 되는지 공부를 했다.

우선 CNN(Convolution Neural Network)에 가장 일반적으로 등장하는 MNIST 문자 인식을 이해하면 원하는 숫자 인식을 쉽게 해결할 수 있다는 결론을 얻어 MNIST를 공부했다(사실 MNIST는 이미 이해하고 알고 있었기 때문에 이 것과 비슷한 구조로 내가 만들어 쓴 것이다).

##### 4.2.1. MNIST

MNIST나 문자 인식을 쉽게 설명한 곳인 <https://tensorflowkorea.gitbooks.io/tensorflow-kr/content/g3doc/tutorials/mnist/beginners/> 를 방문해 우선 이해 하기 바란다. MNIST가 Machine Learning의 "Hello World!"와 같은 것이니 이 내용을 이해 하지 못하면 4.2절의 내용은 이해할 수 없다.

필자는 tensorflow로 공부를 시작했지만 실제 사용한 keras를 선호 한다. keras가 상위 레벨에서 보기 좋게 코딩 할 수 있어서 선호한다(Assembly와 C언어 정도의 차이랄까?).

MNIST 데이터셋은 손으로 쓴 글씨 이미지(28x28) 55,000개 학습용 데이터와 10,000개 테스트용 데이터가 이미 제공 된다. 이 데이터를 받아 사용할 수 있게 만들어 주는 코드는 이미 제공되고 있어 데이터를 마련해야 하는 수고를 덜고 전체가 돌아가는 구조를 쉽게 이해할 수 있다.

그림 7은 MNIST의 CNN(Convolutional Neural Network)을 도식화 한 것이다. 입력 이미지에서 convolution을 거치면서 특징점 들을 찾아 최종 Fully Connected Layer에서 classification을 하는 구조로 되어 있다.

MNIST와 관련된 각종 문서들을 찾아보면 convolution 단계 마다 특징점이 어떻게 찾아지는지 그림으로 표현한 것이 있는데 그 것을 보면 쉽게 이해가 갈 것이다. 특히 마지막의 classification의 softmax 방식의 출력 결과가 무엇인지를 알아 둘 필요가 있다.

출력은 숫자의 경우 0~9까지 10개의 class에 대한 확률 값이다. 즉 출력은 확률값의 array고 총 10개가 출력 되는 것이다. 이 중에 가장 높은 확률 값이 array[5]에 있다면 입력된 이미지는 숫자 6일 가능성이 제일 큰 것이다. 입력 이미지가 부정확하거나 학습이 충분이 이루어지지 않았다면 출력 확률이 비슷한 것이 여러 개 존재할 수도 있다. 이렇게 애매모호한 결과가 나오지 않게 하려면 학습 데이터의 개수와 여러 경우에 대비하는 다양성이 매우 중요하고 그 다음으로는 Network을 어떻게 설계해서 특징점을 충분히 잘 찾을 수 있는가 하는 것이다.

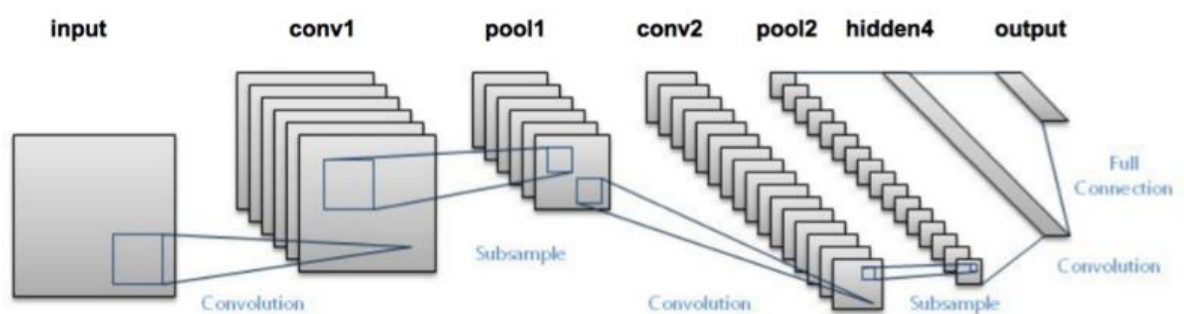


그림 7. Convolutional Neural Network

MNIST의 학습 데이터는 55,000개인데 만약 학습 데이터가 1000개 정도 밖에 안된다고 하면 55,000개에 비해서는 대응 되는 숫자 모양이 적을테고 사람이 손으로 쓰는 경우라면 몇 사람의 필체 밖에는 인식하지 못한다는 소리가 된다. 즉, Machine Learning에서는 입력 되는 학습 데이터의 양이 절대적으로 중요하고 많은 경우에 대비한 충분한 데이터가 필요하다.

애석하게도 MNIST의 목표는 손으로 쓴 글씨를 인식하는 것이고 수도쿠 인식의 경우는 활자(Font)를 사용해 출력된 결과를 카메라로 찍어 인식하는 것이므로 MNIST의 학습 데이터를 사용할 수 없다. 사용 하더라도 활자 출력물과 손으로 쓴 것 사이에 모양의 차이가 심하기 때문에 MNIST로 학습 시킨 것을 활자에 적용하면 인식률이 그리 높지 않은 결과를 갖게 된다.

#### 4.2.2. 입력 데이터

MNIST의 학습 데이터를 사용할 수 없기 때문에 다른 데이터를 사용해 학습해야 하는데 Chars74K 하는 공개 데이터가 있다(<http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/>).

Chars74K 데이터는 문자 인식을 위한 데이터를 제공하는데 다음과 같은 특징을 갖는다.

- 64개 클래스(0-9, A-Z, a-z)
- 이미지로부터 얻은 7705개의 문자 이미지
- 태블릿 PC를 사용해 손으로 쓴 문자 3410개 이미지
- 컴퓨터 폰트를 사용해 만들어진 62992개 문자 이미지

수도쿠는 숫자만 인식하면 되므로 그리고 Font를 사용한 출력물에 대해서만 인식하면 되므로 Chars74K 데이터 중에 컴퓨터 폰트를 사용해 생성된 것중 숫자만 추려 사용한다.

```
train_data = []
truth_data = []

def createData(imgSize=IMAGE_SIZE):
    global train_data, truth_data

    print("Create data: ", end="")
    for i in range(10):
        subDir = 'Sample0{0:02d}'.format(i+1)
        imgDir = os.path.abspath(os.path.join(dataDir, subDir))
        fileList = [f for f in os.listdir(imgDir) if f.endswith('.png')]
        print(i, end="")
        for f in fileList:
            fn = os.path.join(imgDir, f)
            img = Image.open(fn)
            img = img.resize((imgSize, imgSize), Image.ANTIALIAS)
            img = img.convert('RGB')
            data = img.getdata()
            raw = np.array(data, dtype=np.uint8).reshape((imgSize, imgSize, 3))
            train_data.append(raw)
            truth_data.append(i)
```

```

print("")

train_data = np.array(train_data, dtype=np.uint8).reshape((len(train_data), imgSize, imgSize, 3))
train_data.tofile('train_data_{0}.dat'.format(imgSize))
truth_data = np.array(truth_data, dtype=np.int32)
truth_data.tofile('truth_data_{0}.dat'.format(imgSize))

return train_data

```

코드 2. 학습 데이터 생성 코드

코드 2는 Chars74K 데이터의 압축을 풀어 폰트로 만들어진 숫자만 추출한 것에서 내가 원하는 구조의 데이터 파일 하나로 만드는 코드이다.

학습 데이터는 그림 8과 같이 data 디렉토리 밑에 Sample000~Sample009의 디렉토리가 있고 각각의 Sample 디렉토리 밑에는 img001-00001.png와 같은 식의 이름을 가진 128x128 크기의 이미지가 1016개씩 존재한다.

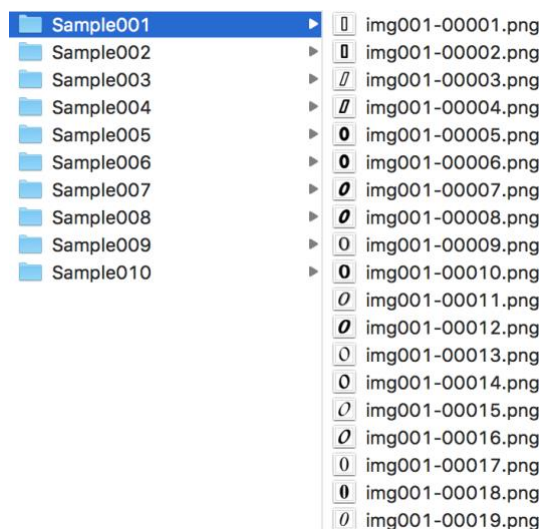


그림 8. 학습 데이터 디렉토리 구조

이 데이터 들을 읽어 raw data만을 추출하고 원하는 크기(예를 들어 64x64)로 resize 한 후 numpy array로 train\_data에 추가 한다. 이와 함께 읽은 이미지 파일이 어떤 숫자인지 참 값을 truth\_data에 추가 한다. 0~9까지의 모든 데이터를 읽어 학습 데이터 train\_data와 참 값인 truth\_data를 만들어 다시 파일로 저장한다. 이렇게 만들어진 데이터는 10,160개의 데이터가 들어 있는 파일이 되고 학습 시에 batch size 만큼씩 이 파일에서 random으로 꺼내 학습을 하게 한다.

수도쿠 이미지에서 숫자 이미지를 잘라내 숫자 인식을 하는 Machine Learning 코드에 입력 해주는 이미지의 크기는 크면 클 수록 좋겠지만 크면 그만큼 속도가 느리기 때문에 그리고 iPhone이라는 최종 디바이스의 성능을 생각하면 되도록 작은 것이 좋을 것이다. 그래서 임의로 64x64 정도의 크기로 정하고 실험을 시작 했다(물론 128x128도 실험을 했으나 64에 비해 나은 점이 별로

없는 듯 해 최종 적으로는 64를 사용했다. 그런데! 더 나중에 보니 잘라낸 이미지를 64x64로 만드니 6, 5, 8, 9 등의 구분이 잘 가지 않는 문제가 있었다. 그래서 128로 좀 더 테스트를 해볼 생각이지만 게을러서 그냥 나중에 생각나면 해볼 것이다...).

#### 4.2.3. 학습

```
import numpy as np

from data import createData, loadData, getBatch
from model import createModel_128, createModel_64

NUM_EPOCH = 10000
NUM_WEIGHTSAVE_PERIOD = 100
BATCH_SIZE = 1024

IMAGE_SIZE = 64

def train(continueTraining=True):
    # load train, truth data
    loadData(imgSize=IMAGE_SIZE)

    # create model
    if IMAGE_SIZE == 128:
        model = createModel_128(inputShape=(IMAGE_SIZE, IMAGE_SIZE, 3), numClasses=10)
    else:
        model = createModel_64(inputShape=(IMAGE_SIZE, IMAGE_SIZE, 3), numClasses=10)
    if continueTraining == True:
        try:
            model.load_weights('weight_{0}.h5'.format(IMAGE_SIZE))
        except:
            print("Weight load exception.")

    # train
    for epoch in range(NUM_EPOCH):
        print("EPOCH={0} : ".format(epoch), end="")
        x_train, y_true = getBatch(numOfBatch=BATCH_SIZE)
        loss = model.train_on_batch(x_train, y_true)
        print('{0:>6.4f} {1:>6.4f}'.format(loss[0], loss[1]), end='Wr')

        if (epoch % NUM_WEIGHTSAVE_PERIOD) == 0:
            model.save('model_{0}.h5'.format(IMAGE_SIZE))
            model.save_weights('weight_{0}.h5'.format(IMAGE_SIZE))
```

```
if __name__ == '__main__':
    train()
```

### 코드 3. 학습 코드

코드 3은 학습을 위한 코드 전체인데 train\_data와 truth\_data 전체를 읽어 들인 후 Model을 생성해 NUM\_EPOCH 회수 만큼 학습을 반복하고 각 학습 때마다 BATCH\_SIZE 만큼의 데이터를 train\_data와 truth\_data에서 random하게 뽑아내어 train\_on\_batch() 함수로 학습을 시킨다.

train\_on\_batch() 함수의 출력은 loss 값과 accuracy 값으로 이루어진 array인데 그 둘을 매 epoch마다 출력해 확인할 수 있도록 했다.

그리고 학습이 진행되는 중간 중간에 Model 과 학습된 weight를 파일로 저장하도록 했다. 이렇게 저장된 Model과 Weight는 나중에 테스트 코드에서 읽어 입력 이미지가 제대로 인식되는지 테스트할 때 사용하고 최종적으로는 iPhone에서 사용 가능한 CoreML Model로 변환해 사용하게 된다.

중간 중간 마다 저장하게 한 이유는 학습이 좋은 GPU 등을 가진 환경에서는 금방 끝나지만 CPU만으로 학습 시키거나 느린 컴퓨터에서 학습을 하면 엄청 오랜 시간이 걸리기 때문에 중간에 저장하게 한 것이다(필자가 사용한 환경은 i7에 GTX1080이었는데 10,000 epoch가 30분 내에 완료되었다).

#### 4.2.4. Model

```
from keras.layers import Input, Conv2D, MaxPool2D, Dropout, Flatten, Dense
from keras.models import Model, load_model

def loadModel(fileName):
    model = load_model(fileName)
    return model

def createModel_128(inputShape=(128, 128, 3), numClasses=10):
    model = createModel_64(inputShape=inputShape, numClasses=numClasses)
    return model

def createModel_64(inputShape=(64, 64, 3), numClasses=10):
    # input
    x = Input(shape=inputShape, name='x')

    y = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
    y = MaxPool2D(pool_size=(2, 2))(y) # 64 -> 32
    y = Conv2D(16, (3, 3), activation='relu', padding='same')(y)
    y = MaxPool2D(pool_size=(2, 2))(y) # 32 -> 16
    y = Conv2D(32, (3, 3), activation='relu', padding='same')(y)
```

```

y = MaxPool2D(pool_size=(2, 2))(y) # 16 -> 8
y = Conv2D(64, (3, 3), activation='relu', padding='same')(y)
y = MaxPool2D(pool_size=(2, 2))(y) # 8 -> 4
y = Conv2D(128, (3, 3), activation='relu', padding='same')(y)
y = MaxPool2D(pool_size=(2, 2))(y) # 4 -> 2
y = Flatten()(y)
y = Dense(512, activation='relu')(y)
y = Dense(numClasses, activation='softmax', name='y')(y)

model = Model(x, y)
model.summary()
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

return model

```

#### 코드 4. Model

MNIST는 28x28x1 크기의 gray 이미지를 입력 받으나 필자가 사용한 것은 64x64x3의 color 이미지를 사용했다. 인식에 있어서 color 이미지가 gray보다 정보가 많으니 유리하기 때문에 이렇게 정의 하긴 했으나 실제 입력되는 이미지가 프로세싱 된 후의 거의 흑백 이미지라서 향후에는 64x64x1 형태로 바뀌야겠다는 생각만 가지고 있다.

Model은 Keras를 사용해 만들었고 5단의 conv + maxpool이 있고 dense가 2단이 있다. 마지막 Dense가 classification을 하는 것으로 512개의 데이터에서 10개의 확률을 만들어 준다.

Model은 adam optimizer를 사용하고 loss의 계산은 categorical\_crossentropy, 출력은 정확도를 출력하게 했다.

loss의 category\_crossentropy 방식은 classification에 적합한 loss 계산이고 학습이 진행되면서 loss 값이 줄어들면 된다.

#### 4.2.5. 테스트

```

from optparse import OptionParser

from data import createData, loadData, getBatch, loadTestData
from model import createModel_64, loadModel

import numpy as np

def test(inputFileName, imgSize=64):
    # load model
    model = loadModel('model_{0}.h5'.format(imgSize))

```



```

# load input data
x_test = loadTestData(inputFileName, imgSize=imgSize)

# predict
pred = model.predict(x_test)
print(pred)

# show result
print('Prediction : {0}'.format(pred[0]))
num = np.argmax(pred[0])
print('Prediction : {0}'.format(num))

if __name__ == '__main__':
    parser = OptionParser()
    parser.add_option("--input", dest="input", default=None, help="Test input")
    parser.add_option("--imgsize", dest="imgsize", type="int", default=64, help="Image size")
    (options, args) = parser.parse_args()

    test(inputFileName=options.input, imgSize=options.imgsize)

```

#### 코드 5. 학습 결과 테스트 코드

테스트는 학습된 Model과 Weight를 읽어 Model을 생성하고 입력 데이터를 입력 포맷에 맞게 읽어 들인 후 predict() 함수를 사용해 인식을 진행한다. predict()의 출력은 10개의 확률 값이고 이 값 중에 제일 큰 값이 인식된 숫자로 제일 큰 값이 들어 있는 array index가 인식된 숫자이다.

학습 당시 정확도가 1.0(=100% 인식 정확도)이었기 때문에 테스트에서는 거의 완벽하게 동작하는 모습을 볼 수 있다.

#### 4.2.6. 그런데...

이렇게 학습된 결과를 iPhone으로 가져가기 전에 생각해 볼 것이 지금 만들고 있는 앱이 사용되는 방식과 인식에 입력되는 이미지가 어떤 모습일 것인가? 하는 것이다.

학습에는 컴퓨터에서 폰트를 사용해 만들어진 숫자 이미지로 학습을 시켰다. 이 이미지는 노이즈도 없고 해상도 손해도 전혀 없는 깨끗한 이미지이지만 실제로 입력 되는 이미지는 카메라로 찍고 resize 등을 통해 추출된 이미지라 노이즈도 있고 흐릿 하기도 하고 특히 수도권 사이트 마다 사용하는 폰트의 이미지가 다르기 때문에 실제 인식과는 거리가 있게 된다.



그림 9. 잘라낸 이미지



그림 10. 폰트로 만든 이미지

그림 9가 카메라로 찍은 이미지에서 찾은 수도쿠 영역을 잘라낸 이미지이다. 노이즈도 있고 흐릿 하기도 하지만 숫자 인식에는 큰 문제 없는 상태이다. 문제는 위와 왼쪽의 수도쿠 칸을 표시한 검은색 선이다. 그리고 이 이미지를 64x64 크기로 resize하면 숫자의 윤곽은 더 흐릿 해진다. 특히 6의 경우 아래 동그라미와 윗 부분이 8에 비해 떨어져 있어야 하는데 카메라로 찍을 때의 환경이나 resize 등의 영향으로 6이 8과 비슷해 보이는 경우가 발생하게 된다.

우선 2가지 문제(검은색 선이 포함되는 문제, 숫자가 흐릿해 보이는 문제)를 해결할 방법을 찾아야 했다.

#### 4.2.6.1. 검은색 선이 포함되는 문제

사실 잘라낸 숫자 이미지에 검은 선이 포함되어 있어도 문제는 없다. 왜? 학습 할 때 사용하는 데이터에 검은색 선이 들어가 있으면 되니까!

학습용 데이터 각 이미지에 4방향에 검은색 선이 조합으로 있을 수 있으므로 총 16가지의 검은색 선이 있고 없는 이미지를 만들어 내면 된다. 그럼 이전에 만든 데이터가 10,160개였으니 x16 해서... 162,560개의 이미지를 만들어 내면 되네?

이미지가 너무 많고 이걸 만드는 것도 별로 나이스!한 방법이 아니라 잘라낸 이미지에서 검은색 선을 제외한 숫자 이미지만 인식해 잘라내 보기로 했다(나중에 보니 그냥 검은색 선이 포함된 이미지를 많이 만드는데 더 좋았을 것이란 생각도 든다).

그림 12는 그림 11을 처리해 숫자 부분만 잘라내 하얀색의 깨끗한 바탕에 넣어준 것이다. Model의 입력에는 이렇게 숫자 부분만 남긴 것을 넘겨 주면 폰트로 만든 깨끗한 학습 데이터 만으로도 충분히 인식률을 높일 수 있어 이 방법을 사용한 것이다.



그림 11. 잘라낸 이미지



그림 12. 숫자만 잘라낸 이미지

#### 4.2.6.2. 숫자만 잘라내 보자

숫자 부분만 잘라내는 코드는 다음과 같다.

```
sz = 128
imageSize = int(sz * 9 / 2)

def getNumImage(sourceMat):
    # gray
    grayMat = cv2.cvtColor(sourceMat, cv2.COLOR_RGB2GRAY)
    # threshold
    threshMat = cv2.adaptiveThreshold(grayMat, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY_INV, 31, 31)
    # find contours
    contourMat, contours, hierarchy = cv2.findContours(
        threshMat,
        cv2.RETR_CCOMP,
        cv2.CHAIN_APPROX_SIMPLE
    )

    contourMat = cv2.cvtColor(contourMat, cv2.COLOR_GRAY2BGR)
    gx = int(sourceMat.shape[1] * 0.05)
    gy = int(sourceMat.shape[0] * 0.05)
    gw = int(sourceMat.shape[1] * 0.9)
    gh = int(sourceMat.shape[0] * 0.9)
    contourMat = cv2.rectangle(contourMat, (gx, gy), (gx+gw, gy+gh), (0, 0, 255), 2)
    # find max contour
    maxArea = 0
    maxContourIndex = -1
    bRect = None
    for i, contour in enumerate(contours):
        area = cv2.contourArea(contour)
        if area > 500:
            x, y, w, h = cv2.boundingRect(contours[i])
            brarea = w * h
```

```

# overrap 영역 찾기
ox = max(gx, x)
oy = max(gy, y)
ox2 = min((gx+gw), (x+w))
oy2 = min((gy+gh), (y+h))
ow = ox2 - ox
oh = oy2 - oy
oarea = ow * oh

# overrap 영역의 크기가 bounding rect와 같으면 couding rect를 포함하고 있는 것
if oarea == brarea:
    # contourMat = cv2.rectangle(contourMat, (x, y), (x+w, y+h), (0, 255, 0), 2)
    # 포함되는 것 중 제일 큰 것만 남긴다
    if area > maxArea:
        maxArea = area
        maxContourIndex = i
        bRect = [x, y, w, h]

# maxContourIndex가 0이상이면 숫자가 있는 것
if maxContourIndex >= 0:
    emptyMat = np.zeros(threshMat.shape, dtype=np.uint8)
    contourMat = cv2.drawContours(contourMat, contours, maxContourIndex, (0, 255, 0), 2)
    x = bRect[0]
    y = bRect[1]
    w = bRect[2]
    h = bRect[3]
    emptyMat[y:y+h, x:x+w] = threshMat[y:y+h, x:x+w]

    # invert color
    numMat = cv2.bitwise_not(emptyMat)
    # resize
    resizedNumMat = cv2.resize(numMat, (sz, sz))
    tmpMat = np.zeros(emptyMat.shape, dtype=np.uint8)
    tmpMat[0:sz, 0:sz] = resizedNumMat

    return cv2.cvtColor(numMat, cv2.COLOR_GRAY2RGB), cv2.cvtColor(resizedNumMat, cv2.COLOR_GRAY2RGB)
else:
    return None, None

```

코드 6. 숫자 부분만 잘라내기 코드

입력 받은 검은색 선이 있는 이미지에서 contour를 찾고 입력 이미지의 95% 크기의 사각형과 겹치는 부분이 있고 그 겹치는 부분을 나타내는 사각형이 현재 contour의 bounding rect와 같으면 95% 안쪽에 있는 것이다. 이 95% 안쪽에 있는 것 중에 제일 큰 contour가 숫자 이미지가 되므로(수도쿠에서 잘라낸 이미지 내부에는 숫자 외에는 없다!) 찾은 숫자 contour 만큼의 이미지를

threshold한 이미지에서 바탕이 검은색인 이미지로 복사한다. 그 후에 이미지를 반전시켜 바탕은 희고 숫자는 검은색이 되도록 만들어 return한다.

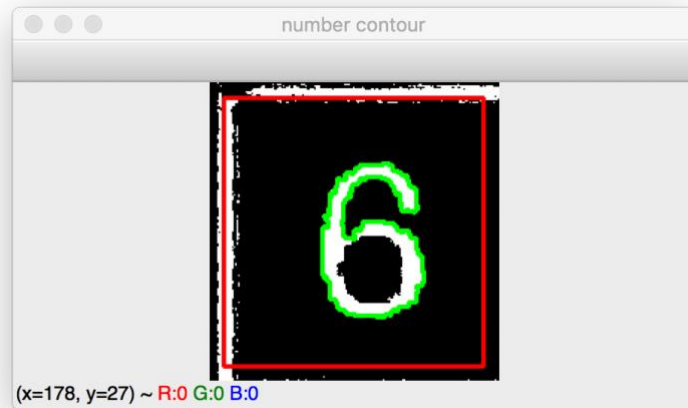


그림 13. 숫자 부분 인식

그림 13에서 빨간색 사각형이 95% 영역을 나타내고 초록색 숫자를 감싼 것이 95% 사각형 안쪽에 존재하는 가장 큰 contour이다. 외곽의 검은색 선은 95% 사각형 안쪽에 위치하지 않기 때문에 제외되었다.

이 방법의 단점이라면 추가 프로세스가 들어가는 것이고 복잡도가 증가하는 것인데 실제로 돌려보니 속도는 걱정하지 않아도 되는 수준이라 그냥 사용했다(내 iPhone이 iPhone X라 빨라서 문제가 없을 수도 있다!).

사실 실제 단점은 threshold를 적용하고 contour를 찾아 복사했기 때문에 원래 이미지에 비해 왜곡된 이미지를 얻는다는 단점이 있다. 그러나 숫자 인식 Model의 입력으로는 문제가 없는 수준이기 때문에 꽤 괜찮은 방법이라 생각된다.

그러나 더 중요한 문제가 발생 했으니 왜곡된 숫자만 찾은 이미지를 폰트로만 만들어진 학습 데이터로 학습한 Model에 넣어주니 인식률이 떨어지는 상황이 감지 되었다. 학습 데이터는 깨끗한 모양인데 입력 되는 숫자 이미지는 울퉁불퉁하고 깨끗하지 않으니 당연한 결과일 수 밖에 없을 것이다. 그래서 학습 데이터에 실제로 얻은 숫자만 찾은 이미지를 추가해 학습을 더 시켜줬고 꽤 만족할 만한 인식률을 얻었다.

#### 4.3. 수도쿠 풀이 알고리즘

수도쿠를 풀어주는 방법은 Wikipedia 등을 찾으면 잘 나와 있는데 간단히 말하면 회귀적으로 한 숫자를 설정해 놓고 다른 셀의 숫자를 맞게 선택한 후 다음 셀로 진행한다. 다음 셀에서 문제가 발생하면 이전 셀로 돌아가 넣어줬던 숫자 말고 다른 숫자를 넣어 다시 진행하는 식으로 해결을

하는 것이다.

[https://en.wikipedia.org/wiki/Sudoku\\_solving\\_algorithms](https://en.wikipedia.org/wiki/Sudoku_solving_algorithms) 에 있는 애니메이션을 보면 쉽게 이해가 갈 것이고 이 방법 대로 구현 해보면 이해가 될 것이다.

코드 7은 Wikipedia의 알고리즘을 검증하기 위해 초기에 C#으로 짰던 코드이다. 알고리즘을 이해하고 쓰지는 않았다(화면 표시를 쉽게하는 방법에 대해 윈도우즈에서 하는게 편하고 쉬워서 C#으로 짰 것이다. Python은 curses 등을 써야하는데 오히려 귀찮아서...).

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Threading;

namespace WindowsFormsApp1
{
    public partial class Form1 : Form
    {
        public struct SudokuCell
        {
            public int number;
            public bool reserved;
            public bool valid;

            public SudokuCell(int n, bool r, bool v)
            {
                number = n;
                reserved = r;
                valid = v;
            }
        }

        private EventWaitHandle _waitForSingleSignal;

        private const int MAX_ROW = 9;
        private const int MAX_COL = 9;
```

```

private SudokuCell[][] _sudokuMatrix;

public Form1()
{
    InitializeComponent();
    _waitForSingleSignal = new EventWaitHandle(false, EventResetMode.AutoReset);
    DoubleBuffered = true;
}

private void button1_Click(object sender, EventArgs e)
{
    _sudokuMatrix = _generateSudokuMatrix();
    _solveSudoku(_sudokuMatrix);
}

private void button2_Click(object sender, EventArgs e)
{
    _waitForSingleSignal.Set();
}

private SudokuCell[][] _generateSudokuMatrix()
{
    int[,] val = new int[,] {
        {5, 3, 0, 0, 7, 0, 0, 0, 0},
        {6, 0, 0, 1, 9, 5, 0, 0, 0},
        {0, 9, 8, 0, 0, 0, 0, 6, 0},
        {8, 0, 0, 0, 6, 0, 0, 0, 3},
        {4, 0, 0, 8, 0, 3, 0, 0, 1},
        {7, 0, 0, 0, 2, 0, 0, 0, 6},
        {0, 6, 0, 0, 0, 0, 2, 8, 0},
        {0, 0, 0, 4, 1, 9, 0, 0, 5},
        {0, 0, 0, 0, 8, 0, 0, 7, 9}
    };

    //int[,] val = new int[,] {
    //    {7, 0, 1, 3, 0, 9, 0, 0, 5},
    //    {0, 0, 0, 0, 0, 1, 6, 0, 4},
    //    {5, 0, 4, 0, 0, 0, 7, 0, 0},
    //    {0, 0, 0, 0, 0, 0, 0, 1, 0},
    //    {8, 0, 7, 6, 0, 4, 9, 0, 0},
    //    {0, 1, 0, 5, 0, 0, 3, 0, 6},
    //    {9, 0, 3, 0, 6, 5, 0, 8, 7},
    //    {0, 5, 0, 0, 0, 0, 1, 0, 0},
    //    {0, 0, 0, 8, 3, 0, 0, 0, 9}
    //};

```

```

    //};

    //int[,] val = new int[,] { // 고난이도
    //    {1, 0, 0, 0, 0, 0, 0, 0, 3},
    //    {0, 0, 0, 0, 6, 0, 0, 0, 0},
    //    {0, 0, 3, 0, 0, 1, 0, 0, 0},
    //    {0, 7, 0, 1, 0, 0, 0, 0, 0},
    //    {0, 0, 8, 0, 0, 0, 5, 0, 0},
    //    {0, 0, 0, 0, 0, 3, 0, 4, 0},
    //    {0, 0, 0, 8, 0, 0, 6, 0, 0},
    //    {0, 0, 0, 0, 1, 0, 0, 0, 0},
    //    {6, 0, 0, 0, 0, 0, 0, 0, 7},
    //};

    //int[,] val = new int[,] {
    //    {5, 0, 0, 0, 1, 0, 0, 0, 4},
    //    {2, 7, 4, 0, 0, 0, 6, 0, 0},
    //    {0, 8, 0, 9, 0, 4, 0, 0, 0},
    //    {8, 1, 0, 4, 6, 0, 3, 0, 2},
    //    {0, 0, 2, 0, 3, 0, 1, 0, 0},
    //    {7, 0, 6, 0, 9, 1, 0, 5, 8},
    //    {0, 0, 0, 5, 0, 3, 0, 1, 0},
    //    {0, 0, 5, 0, 0, 0, 9, 2, 7},
    //    {1, 0, 0, 0, 2, 0, 0, 0, 3},
    //};

    List<List<SudokuCell>> sm = new List<List<SudokuCell>>();
    for (int row = 0; row < 9; row++)
    {
        List<SudokuCell> sm_row = new List<SudokuCell>();
        for (int col = 0; col < 9; col++)
        {
            SudokuCell sc;

            if (val[row, col] != 0)
                sc = new SudokuCell(val[row, col], true, true);
            else
                sc = new SudokuCell(0, false, false);

            sm_row.Add(sc);
        }
        sm.Add(sm_row);
    }

    return sm.Select(a => a.ToArray()).ToArray();

```



```

}

private bool _solveSudoku(SudokuCell[][] sudokuMatrix)
{
    for (int row = 0; row < MAX_ROW; row++)
    {
        for (int col = 0; col < MAX_COL; col++)
        {
            SudokuCell sc = sudokuMatrix[row][col];
            if (sc.reserved == true)
            {
                // reserved임 다음 cell로 이동
                continue;
            }
            else
            {
                int validNumber;
                if (sc.valid == true)
                {
                    // 현재 cell이 valid면 뭔가 다음 cell에서 valid를 못 찾는 경우임
                    // 현재 cell의 valid number를 새로 찾는다
                    validNumber = _findNextValidNumber(sudokuMatrix, row, col,
sudokuMatrix[row][col].number + 1);
                }
                else
                {
                    // 현재 cell은 빈 cell이므로 1부터 시작해 valid를 찾는다
                    validNumber = _findNextValidNumber(sudokuMatrix, row, col, 1);
                }

                if (validNumber < 0)
                {
                    // valid를 못 찾음, 이전 cell로 이동
                    sudokuMatrix[row][col] = new SudokuCell(0, false, false);
                    int num = _findPrevValidCell(sudokuMatrix, ref row, ref col);
                    if (num > 0)
                    {
                        // 찾음
                        col--;
                    }
                    else
                    {
                        // 못찾음

```

```

        return false;
    }
}
else
{
    // valid 찾음
    sudokuMatrix[row][col] = new SudokuCell(validNumber, false, true);
    // 다음 cell로 이동
}
}
_display(sudokuMatrix);
//System.Threading.Thread.Sleep(500);
}
//_waitForSingleSignal.WaitOne();
}
return false;
}

private int _findPrevValidCell(SudokuCell[][] sudokuMatrix, ref int startRow, ref int startCol)
{
    int scol = startCol - 1;
    for (int row = startRow; row >= 0; row--)
    {
        for (int col = scol; col >= 0; col--)
        {
            SudokuCell sc = sudokuMatrix[row][col];
            if (sc.reserved == true)
                continue;
            else
            {
                startRow = row;
                startCol = col;
                return sc.number;
            }
        }
        scol = MAX_COL - 1;
    }
    return -1;
}

private int _findNextValidNumber(SudokuCell[][] sudokuMatrix, int row, int col, int start_num)
{
    for (int num = start_num; num < 10; num++)

```

```

    {
        if (_checkRowValid(sudokuMatrix, row, num) == true &&
            _checkColValid(sudokuMatrix, col, num) == true &&
            _checkLocalValid(sudokuMatrix, row, col, num) == true)
        {
            // 3가지 조건 모두 만족
            return num;
        }
        else
        {
            // valid 하지 않음
            continue;
        }
    }

    // 모든 숫자가 다 만족하지 않음
    return -1;
}

private bool _checkRowValid(SudokuCell[][] sudokuMatrix, int row, int number)
{
    for (int col = 0; col < MAX_COL; col++)
    {
        if (sudokuMatrix[row][col].number == number)
        {
            // 같은 숫자가 존재
            return false;
        }
    }

    // 같은 숫자 존재하지 않음
    return true;
}

private bool _checkColValid(SudokuCell[][] sudokuMatrix, int col, int number)
{
    for (int row = 0; row < MAX_ROW; row++)
    {
        if (sudokuMatrix[row][col].number == number)
        {
            // 같은 숫자가 존재
            return false;
        }
    }

    // 같은 숫자 존재하지 않음

```

```

        return true;
    }

    private bool _checkLocalValid(SudokuCell[][] sudokuMatrix, int row, int col, int number)
    {
        int startRow = (int)(row / 3) * 3;
        int startCol = (int)(col / 3) * 3;

        for (int r = startRow; r < (startRow + 3); r++)
        {
            for (int c = startCol; c < (startCol + 3); c++)
            {
                if (sudokuMatrix[r][c].number == number)
                {
                    // 같은 숫자가 존재
                    return false;
                }
            }
        }

        // 같은 숫자 존재하지 않음
        return true;
    }

    private void _display(SudokuCell[][] sudokuMatrix)
    {
        Bitmap b = new Bitmap(pictureBox1.ClientSize.Width, pictureBox1.ClientSize.Height);
        Graphics g = Graphics.FromImage(b);
        g.FillRectangle(Brushes.White, pictureBox1.ClientRectangle);

        Font f = new Font("D2Coding", 10f);

        float x = 0f;
        float y = 0f;

        for (int row = 0; row < MAX_ROW; row++)
        {
            for (int col = 0; col < MAX_COL; col++)
            {
                SudokuCell sc = sudokuMatrix[row][col];

                if (sc.reserved == true)
                {
                    g.DrawString(sc.number.ToString(), f, Brushes.Black, new PointF(x, y));
                }

                else if (sc.number != 0 && sc.valid == true)
            }
        }
    }

```

```

        {
            g.DrawString(sc.number.ToString(), f, Brushes.Red, new PointF(x, y));
        }
        else if (sc.number != 0)
        {
            g.DrawString(sc.number.ToString(), f, Brushes.Green, new PointF(x, y));
        }
        x += 10f;
    }
    x = 0f;
    y += 12f;
}
g.Flush();
g.Dispose();

g = pictureBox1.CreateGraphics();
g.DrawImage(b, new Point(0, 0));
g.Dispose();
}
}
}

```

코드 7. 수도쿠 풀이 알고리즘 확인 코드

실제로 사용한 코드는 <https://spin.atomicobject.com/2012/06/18/solving-sudoku-in-c-with-recursive-backtracking/> 에서 구현한 방법을 따라 Python으로 구현해 확인하고 마지막에 iPhone 의 swift로 코딩을 해서 사용했다.

```

import datetime

# number가 현재 cell에 맞는지 검사
def isValid(number, sudoku, row, col):
    i = 0
    sectorRow = 3 * int(row / 3)
    sectorCol = 3 * int(col / 3)
    row1 = (row + 2) % 3
    row2 = (row + 4) % 3
    col1 = (col + 2) % 3
    col2 = (col + 4) % 3

    # number가 row, column에 존재하는지 검사
    for i in range(9):
        if sudoku[i][col] == number:

```

```

        return False

    if sudoku[row][i] == number:
        return False

    # cell이 속한 section의 나머지 4 귀퉁이 cell에도 같은 숫자가 있는지 검사
    if sudoku[row1 + sectorRow][col1 + sectorCol] == number:
        return False
    if sudoku[row2 + sectorRow][col1 + sectorCol] == number:
        return False
    if sudoku[row1 + sectorRow][col2 + sectorCol] == number:
        return False
    if sudoku[row2 + sectorRow][col2 + sectorCol] == number:
        return False

    # 모든 것을 통과! valid!
    return True

def sudoku_solver(sudoku, row, col):
    if row == 9:
        return True

    # cell에 이미 숫자가 있는 경우는 변경하지 않고 바로 다음 cell로 넘어감
    if sudoku[row][col] != 0:
        if col == 8:
            if sudoku_solver(sudoku, row+1, 0) == True:
                return True
        else:
            if sudoku_solver(sudoku, row, col+1) == True:
                return True
        return False

    # iteration을 돈다
    for nextNum in range(1, 10):
        if isValid(nextNum, sudoku, row, col) == True:
            sudoku[row][col] = nextNum
            if col == 8:
                if sudoku_solver(sudoku, row+1, 0) == True:
                    return True
            else:
                if sudoku_solver(sudoku, row, col+1) == True:
                    return True
    # 이 cell에 대해 valid value를 찾지 못함
    sudoku[row][col] = 0

```

```

if __name__ == '__main__':
    sudoku_1 = [
        [0, 0, 1, 0, 4, 5, 0, 9, 0],
        [0, 0, 0, 1, 0, 2, 0, 0, 0],
        [0, 0, 4, 3, 0, 0, 0, 7, 0],
        [6, 1, 2, 0, 0, 0, 0, 0, 0],
        [0, 3, 0, 0, 0, 0, 0, 8, 0],
        [0, 0, 0, 0, 0, 0, 3, 5, 6],
        [0, 2, 0, 0, 0, 1, 8, 0, 0],
        [0, 0, 0, 7, 0, 6, 0, 0, 0],
        [0, 7, 0, 9, 8, 0, 6, 0, 0]
    ]

    start = datetime.datetime.now()
    sudoku_solver(sudoku_1, 0, 0)
    end = datetime.datetime.now()
    print('elapsed time = {}'.format(end - start))
    print(sudoku_1)

```

**코드 8. 실제 사용된 수도쿠 풀이 코드**

Recursive Backtracking 방식을 사용한 것이고 코드가 아주 간결한 것을 볼 수 있다. isValid와 sudoku\_solver 함수 2개로 구성되어 있고 sudoku\_solver 함수가 recursive 하게 호출되어 해결하는 것이다.

이 코드를 돌려보면 C#으로 짰던 코드와 개념은 같지만 훨씬 속도가 빠른 것을 볼 수 있다. 특히 올바른 입력에 대해서는 0.x 초만에 답을 찾아 준다.

그러나 반대로 숫자 인식이 틀려 올바르지 않은 수도쿠가 입력되면 recursive 호출에서 빠져 나오지 못하는 경우도 발생한다. 아직 이에 대한 처리는 하지 않았는데 문제가 발생했다는 것을 특정 지을 명확한 기준을 세우지 못해 나중에 미뤄 두었다(사실 회사 출장 준비로 바빠 얼마간 멈춰있더니 하기 싫어져서...^^).

#### 4.4. iPhone에 Machine Learning 결과 돌리기

보통 Machine Learning은 Python으로 코딩을 한다. Tensorflow도 기본적으로 Python에서 코딩하기가 제일 좋다. 또 Python 하에서 Machine Learning을 공부하거나 제작하는 사람이 많으니 점점 더 활성화 되는 positive feedback 효과도 있을 것이다.

하여튼 숫자 인식을 Python으로 작성해 학습시키고 테스트해서 원하는 결과를 얻었는데 이것 iPhone에 어떻게 넣어줄 것인가? iPhone이 Python 기반이라면 바로 사용하면 되겠지만 최근엔 Swift로 코딩을 하고 SDK도 완전 다르기 때문에 Machine Learning의 결과물을 사용할 방법을 찾아야만 한다.

#### 4.4.1. CoreML Model 변환

iPhone하에서는 (엄밀히 말하면 Apple 디바이스 코딩 환경 하에서는) CoreML이라는 Framework 을 사용해 Machine Learning 결과물을 사용한다.

<https://developer.apple.com/documentation/coreml> 을 보면 CoreML에 대한 설명이 있으니 무엇인지 정도는 알아두는 것이 좋다.

Python으로 만든 Machine Learning을 CoreML에서 사용하는 방법은 보통 다음과 같다.

- ① Python으로 코딩, 학습 데이터 준비
- ② 학습 진행
- ③ 학습된 Model과 Weight 저장
- ④ 저장된 Model과 Weight를 CoreML Model로 변환
- ⑤ XCode project에 변환된 CoreML Model 등록
- ⑥ 등록된 CoreML Model 사용

중요한 것은 변환과 실제 사용해 동작을 확인하는 것이므로 우선 학습된 Model을 변환 하는 방법에 대해 알아보았다.

변환은 python pip로 coremltools라는 모듈을 설치하고 진행한다. 위에서도 언급한 것처럼 보통은 Python으로 Model을 만들기 때문에 Python에서 CoreML로의 변환 tool이 제공되는 것이다.

```
from optparse import OptionParser
import coremltools
import os
from model import createModel_128, createModel_64

def convert(inputModel=None):
    if inputModel == None:
        print('No model was specified.')
        return

    fn, ext = os.path.splitext(inputModel)

    """ create model and convert
    # model = createModel_64(inputShape=(64, 64, 3), numClasses=10)
    # coreml_model = coremltools.converters.keras.convert(model)
    """

    # load saved model and convert
```



```

coreml_model = coremltools.converters.keras.convert(
    inputModel,
    input_names=['x'],
    output_names=['y'],
    image_input_names=['x'],
    image_scale=1/255.0
)
coreml_model.save(fn + '.mlmodel')

'''
# minimize model
coreml_model_fp16_spec = coremltools.utils.convert_neural_network_spec_weights_to_fp16(coreml_model)
coreml_model_fp16_spec.save(fn + '_fp16.mlmodel')
coremltools.utils.save_spec(coreml_model_fp16_spec, fn + '_fp16.mlmodel')
'''

if __name__ == '__main__':
    parser = OptionParser()
    parser.add_option("--input", dest="input", default=None)
    (options, args) = parser.parse_args()

    convert(options.input)

```

### 코드 9. CoreML 변환 코드

Machine Learning Model은 train.py에서 'model\_64.hd5'라는 이름으로 저장된다. Keras에서 model을 저장하면 학습된 weight까지 같이 저장되므로 weight를 따로 저장하거나 불러 사용할 필요는 없다. 필자는 Keras로 Model을 만들었기 때문에 coremltools의 변환 기능 중에 keras.convert()를 사용한 것이다. 다른 것으로 만들었다면 그에 맞게 변환 하는 방법을 coremltools 문서를 통해 확인 바란다.

convert() 함수에서 가장 중요한 것은 image\_input\_names 파라미터를 지정해 주는 것이다. 이 파라미터를 지정해 주지 않으면 이미지를 입력 받지 않고 다른 형식의 데이터를 입력 받아 어떻게 써야하는지 알기 힘들어진다. **매우 매우 중요한 파라미터이다.**

그리고 image\_scale 파라미터는 입력되는 이미지가 RGB 8bit 값으로 이루어져 있다면 그 각각의 RGB 값을 0~1.0 사이의 값으로 normalize 해주는 것이다. 실제로 train.py의 loadData() 함수에도 학습 이미지를 batch로 만들어줄 때 1/255를 해준다. 이렇게 하지 않으면 계산 할 때 값이 커져 발산하기 쉬워 학습이 제대로 안될 수 있다. 그러므로 normalize 해주는 것이 좋다(실험적으로 학습할 때 1/255를 빼보면 학습이 제대로 안되는 것을 볼 수 있다).

변환된 CoreML Model은 반드시 '.mlmodel' 확장자를 갖도록 해줘야 한다. 그래야 XCode에서 자동인식되고 쉽게 사용할 수 있다.

변환 코드 중에 사용되지 않는 minimize model 부분이 있는데 Python에서 만들어진 Model은 계산 할 때 full-precision float를 기반으로 하기 때문에 PC에서는 쉽고 빠르게 동작하나 iPhone에서 못 돌리는 것은 아니나 용량을 줄이기 위해서 half-precision으로 변환 하는 것이 좋다. iPhone의 메모리는 제약이 심하기 때문에 PC에서 만큼 맘대로 사용하면 앱이 죽어 버리는 경우가 빈번히 발생한다.

그러나 뭘 일인지 half-precision 변환 코드가 예러가 나서 동작을 하지 않아 comment out 해 놓은 것이다(나중에 알아봐야지 하면서 아직도 못 알아봤네요...).

#### 4.4.2. XCode에서 사용

변환된 CoreML Model을 XCode project에 drag&drop 해주면 된다. XCode가 자동으로 인식해 코드에서는 Class로 사용할 수 있게 해준다.

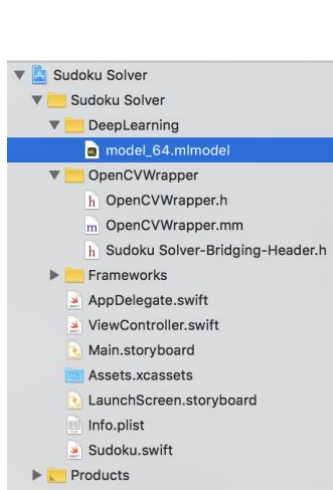


그림 14. XCode에 CoreML Model 등록

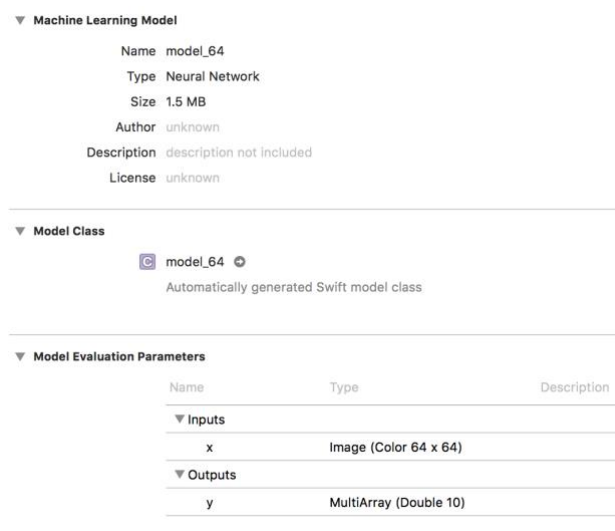


그림 15. 등록된 CoreML Model 정보

등록된 CoreML Model를 클릭하면 그 정보를 확인할 수 있는데 Inputs에 'Image(Color 64x64)'라는 것을 볼 수 있을 것이다. 위의 변환에서 설명할 것처럼 image\_input\_names 파라미터를 지정해 주었기 때문에 입력으로 이미지를 받는 것이다.

입력에 대해 Python에서는 우리가 원하는 만큼(batch size)의 입력 이미지를 줄 수 있었는데 CoreML에서는 입력이 1개로 제한된다. 그러므로 수도쿠 숫자 인식에서 81개의 모든 이미지를 넣어주는게 아니라 1개씩 처리 해야한다.

출력은 인식된 10개의 클래스에 대한 확률 값이다. MultiArray라는 형식으로 되어 있는데 이게 사용하기 까다로웠다. 결국은 stackoverflow.com을 뒤져야만 했다.

그리고 CoreML Model 정보 창에서 Model Class 항목에 있는 model\_64 옆의 화살표를 클릭하면 model\_64.swift라는 코드를 볼 수 있다. 이 코드가 XCode project에 등록할 때 자동으로 생성된

것으로 model\_64라는 class를 생성해 사용할 수 있도록 해준다.

#### 4.4.3. CoreML Model 사용

등록된 CoreML Model은 class로 쓸 수 있게 되므로 let model = model\_64()라는 코드로 class를 생성해 쓴다.

입력 이미지는 CVPixelBuffer 형식으로 만들어야 하고 model.prediction() 함수가 인식을 실행해 준다.

입출력의 이름을 지정해 줬었기 때문에 .x, .y를 사용할 수 있는 것이다.

출력의 10개 double 값을 알아내는 방법은 prediction 밑의 코드 들이고 let output = Array(doubleBuffer) 코드에서 10개의 double 값 Array을 얻을 수 있게 된다. 여기에서 값이 제일 큰 것이 인식된 숫자이므로 그 index가 인식된 숫자 값이다.

```
// 숫자가 존재 하는 경우 처리
let buf = img.pixelBuffer()

// predict
let model = model_64()
guard let pred = try? model.prediction(x: buf!) else {
    print("prediction exception")
    break
}

let length = pred.y.count
let doublePtr = pred.y.dataPointer.bindMemory(to: Double.self, capacity: length)
let doubleBuffer = UnsafeBufferPointer(start: doublePtr, count: length)
let output = Array(doubleBuffer)
let maxVal = output.max()
let maxIdx = output.index(of: maxVal!)
sudokuArray[row][col] = maxIdx!
```

코드 10. CoreML Model 사용 코드

## 5. iPhone 앱 제작

### 5.1. 동작 구상

아내가 쓰는 것을 목표로 했으므로 그리고 나에게 더 이상의 질문이 오면 안되고 알아서 잘 사용할 수 있으려면 어떻게 만들어야 할까? 직관적으로 동작 시킬 수 있어야 하고 조작이 되도록 적어야 한다. 또 수도쿠를 카메라로 찍어야 하니 카메라로 찍을 때 guide가 있어야 할 것이다. 찍은

수도쿠가 제대로 인식 됐는지 알 수 있으면 좋겠고 제대로 인식 안되면 간단히 다시 시도할 수 있으면 좋겠다.

그래서 그림 16과 같이 UI 디자인을 했다.

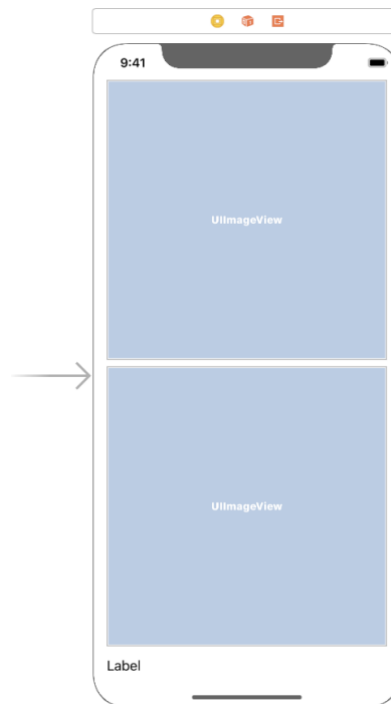


그림 16. UI 디자인

위쪽 UIImageView는 카메라로 수도쿠를 찍을 때 preview를 보여주는 것이고 정사각형으로 만들었는데 수도쿠를 최대한 이 UIImageView에 들어가도록 찍기를 유도한 것이다.

아래쪽 UIImageView는 두 가지 동작을 하는데 첫 번째는 터치 입력을 받아 카메라 preview에서 사진을 찍고 풀이를 진행하고 카메라 촬영과 동시에 카메라 preview를 pause 시켜둔다. 두 번째 역할은 카메라 preview에서 입력되는 이미지에서 수도쿠 영역을 실시간으로 찾아 보여주거나 풀이된 수도쿠 결과를 보여주는 역할을 한다.

일단 풀이가 되거나 오류가 있었던 경우 아래 UIImageView를 다시 터치하면 카메라 preview가 다시 동작하고 사진 촬영을 할 수 있는 상태로 간다.

이렇게 조작은 아래 부분을 터치하는 동작으로만 모든 것을 할 수 있고 찍을 때 최대한 정사각형 안에 크게 찍기만 하면 된다. (이 정도면 한번 해보면 직관적으로 사용할 수 있겠지?)

Label은 디버깅 정보 등을 보기 위해 넣어 놨는데 현재는 걸린 시간만 표시 한다.

## 5.2. 카메라 동작

iPhone에서 카메라를 동작시키는 방법을 알아야 하는 것이 가장 먼저라 문서, 예제 등을 찾아 코딩을 했다.

### 5.2.1. 카메라 준비

```
override func viewDidLoad() {
    super.viewDidLoad()

    // processedImageView가 touch 됐을 때의 처리
    processedImageView.isUserInteractionEnabled = true
    processedImageView.addGestureRecognizer(UITapGestureRecognizer(target: self, action:
#selector(self.processedImageViewTouched(_:))))

    // camera capture를 준비
    prepareCapture()
    // camera capture 시작
    startCapture()
}

func prepareCapture() {
    // setup device
    let captureDevice = AVCaptureDevice.default(for: AVMediaType.video)
    do {
        // setup input
        let input = try AVCaptureDeviceInput(device: captureDevice!)

        // setup session
        captureSession = AVCaptureSession()
        captureSession?.sessionPreset = AVCaptureSession.Preset.hd1280x720
        captureSession?.addInput(input)

        // setup output
        let videoOutput = AVCaptureVideoDataOutput()
        videoOutput.videoSettings = [kCVPixelBufferPixelFormatTypeKey as AnyHashable as! String: NSNumber(value:
kCVPixelFormatType_32BGRA)]

        // setup session queue
        let sessionQueue = DispatchQueue(label: "VideoQueue")
        videoOutput.setSampleBufferDelegate(self, queue: sessionQueue)
        captureSession?.addOutput(videoOutput)

        // setup preview layer
```

```

        videoPreviewLayer = AVCaptureVideoPreviewLayer(session: captureSession!)
        videoPreviewLayer?.videoGravity = AVLayerVideoGravity.resizeAspectFill
        videoPreviewLayer?.connection?.videoOrientation = AVCaptureVideoOrientation.portrait
        videoPreviewLayer?.frame = cameraImageView.layer.bounds
        cameraImageView.layer.addSublayer(videoPreviewLayer!)
    } catch {
        print(error)
    }
}

func startCapture() {
    captureSession?.startRunning()
}

func stopCapture() {
    captureSession?.stopRunning()
}

```

코드 11. 카메라 동작 코드

viewDidLoad()에서 카메라 동작을 준비해 놓고 startCapture()로 preview에 카메라 화면이 나오도록 하고 stopCapture()로 pause하는 것이다.

카메라 출력은 AVCaptureSession.Preset.hd1280x720를 세팅해 1280x720 크기의 이미지를 출력하게 한다. 그리고 DispatchQueue를 사용해 async 하게 동작할 수 있도록 했다.

### 5.2.2. 카메라 출력 처리

class ViewController의 선언에 AVCaptureVideoDataOutputSampleBufferDelegate를 추가하면 ViewController 내에 captureOutput() 함수를 선언해 사용할 수 있게 된다.

카메라 동작 준비 동안 카메라의 출력에 setSampleBufferDelegate()를 사용해 self를 지정했기 때문이다.

```

func captureOutput(_ output: AVCaptureOutput, didOutput sampleBuffer: CMSampleBuffer,
    from connection: AVCaptureConnection) {
    // 화면 회전에 따라 얻어지는 이미지의 방향도 바꿔준다
    connection.videoOrientation = AVCaptureVideoOrientation(rawValue: UIDevice.current.orientation.rawValue)!

    // image 얻기
    let imageBuffer = CMSampleBufferGetImageBuffer(sampleBuffer)!
    CVPixelBufferLockBaseAddress(imageBuffer, CVPixelBufferLockFlags(rawValue: CVOptionFlags(0)))

    let width = CVPixelBufferGetWidth(imageBuffer)

```

```

let height = CVPixelBufferGetHeight(imageBuffer)
let bitsPerComponent = 8
let bytesPerRow = CVPixelBufferGetBytesPerRow(imageBuffer)
let baseAddress = CVPixelBufferGetBaseAddress(imageBuffer)!

let colorSpace = CGColorSpaceCreateDeviceRGB()
let bitmapInfo = CGLImageAlphaInfo.premultipliedFirst.rawValue | CGBitmapInfo.byteOrder32Little.rawValue
let newContext = CGContext(data: baseAddress, width: width, height: height, bitsPerComponent: bitsPerComponent,
bytesPerRow: bytesPerRow, space: colorSpace, bitmapInfo: bitmapInfo)
if let context = newContext {
    let cameraFrame = context.makeImage()
    DispatchQueue.main.async {
        // create UIImage
        let img = UIImage(cgImage: cameraFrame!)
        // crop
        let w = img.size.width
        let y = (img.size.height - w) / 2
        let r = CGRect(x: 0, y: y, width: w, height: w)
        let imgRef = img.cgImage?.cropping(to: r)
        let capturedImage = UIImage(cgImage: imgRef!)

        self.doGetSudokuRect(capturedImage)
    }
}

CVPixelBufferUnlockBaseAddress(imageBuffer, CVPixelBufferLockFlags(rawValue: CVOptionFlags(0)))
}

```

## 코드 12. 카메라 출력 처리

코드 12의 첫부분 화면 회전에 대한 코드는 나중에 설명하기로 하고... 카메라의 출력이 생길 때 마다 `captureOutput()`이 불리는데 카메라로부터 이미지를 얻어 수도쿠 영역을 찾아내는 함수를 `DispatchQueue`를 사용해 `async`하게 실행 해준다.

`async`하게 실행하는 이유는 수도쿠 영역 찾는 작업이 얼마나 오래 걸리는지도 모르고 카메라 출력이 30Hz 정도만 되도 수도쿠 영역 찾는 작업을 같이 실행해 버리면 다음 번 출력을 놓치거나 밀려 처리할 수 밖에 없게 된다. 이런 일을 막기 위해 출력을 받는 함수에서는 출력을 받기만 하고 `DispatchQueue`를 사용해 뒤의 처리는 나중에 하도록 넘기는 것이다(동작 시켜보면 알겠지만 iPhoneX에서 수도쿠 영역 검출은 시간이 얼마 걸리지 않아 `DispatchQueue`를 사용하지 않아도 충분히 잘 동작한다).

수도쿠 영역을 찾는 것은 `doGetSudokuRect()` 함수인데 이 함수를 호출하기 전에 카메라의 출력이 1280x720 크기로 설정 됐지만 `preview`는 정사각형으로 되어 있으므로 1280x720 크기에서 정사각형 부분만 잘라내 수도쿠 영역 검출 함수로 넘겨주는 것이다. `preview`의 영역은 1280x720의

중앙에 위치하기 때문에 그리고 좌우 넓이는 preview에 그대로 다 보이므로 세로에 해당하는 높이만 잘라주면 된다. 실제 잘라내는 이미지는 720x720 크기가 된다.

### 5.3. 아래 UIImageView 터치 이벤트 처리

5.1에서 언급한 것과 같이 아래쪽 UIImageView를 터치하면 카메라 preview가 멈추고 수도쿠 인식을 한다. 이 때 카메라 preview는 pause 된다. 이 상태에서 다시 터치하면 카메라 preview가 다시 동작하는 상태로 돌아간다.

```
@objc func processedImageViewTouched(_ sender: UITapGestureRecognizer) {
    if pauseStatus == false {
        // 현재 camera capture 중인 상태
        pauseStatus = true
        stopCapture()

        // sudoku 풀이 queue 생성
        sudokuSolvingWorkItem = DispatchWorkItem(block: self.sudokuSolvingQueue)
        sudokuSolvingStart = NSDate()
        DispatchQueue.main.async(execute: sudokuSolvingWorkItem!)
    } else {
        // 현재 camera capture 멈춘 상태
        // camera capture를 다시 시작
        pauseStatus = false
        startCapture()

        infoLabel.text = ""
    }
}
```

코드 13. 터치 이벤트 처리

UIImageView는 기본적으로 터치 이벤트를 처리하지 못하게 되어 있어 viewDidLoad() 함수에서 다음과 같이 터치 이벤트 처리 함수를 등록 해줘야 한다.

```
// processedImageView 가 touch 됐을 때의 처리
processedImageView.isUserInteractionEnabled = true
processedImageView.addGestureRecognizer(UITapGestureRecognizer(target: self, action:
#selector(self.processedImageViewTouched(_:))))
```

코드 14. 터치 이벤트 등록

isUserInteractionEnabled를 true로 해줌으로써 이벤트 처리를 할 수 있게 해주고 UITapGestureRecognizer를 등록해 터치 됐을 때 processedImageViewTouched()가 불리게 해준다. UITapGestureRecognizer를 생성할 때 action에는 selector로 등록을 하는데 이 함수는 @objc라는



attribute를 가져야만 한다.

터치 이벤트가 발생하면 현재의 pauseStatus 변수를 확인해 카메라가 동작 중인 상태라면 카메라를 멈추고 수도쿠 풀이를 진행한다. 수도쿠 풀이를 위한 수도쿠 영역 이미지는 카메라가 동작하는 동안 아래 UIImageView에 이미 들어 있기 때문에 수도쿠 풀이하는 함수에서 이 이미지를 가져다 숫자 인식 후 풀이를 진행한다.

#### 5.4. 수도쿠 풀이

```
func doSudokuSolving(sudokulImage: UIImage) {
    // get sudoku number images
    var sudokuArray:[[Int]] = Array(repeating: Array(repeating: 0, count: 9), count: 9)
    if let r2 = OpenCVWrapper.getSlicedSudokuNumImages(sudokulImage, imageSize: 64, cutOffset: 0) {
        // r2[0]는 sudoku 영역을 9x9로 자르고 각각의 이미지를 64x64 크기로 변환한 UIImage array
        // r2[1]은 디버깅 목적의 9x9로 자른 이미지를 다시 하나에 합쳐 놓은 이미지(제대로 잘렸는지 보기 위한 용도)

        let numImages = r2[0] as! NSArray
        let predStartTime = NSDate()
        for i in 0..
```

```

        let maxIdx = output.index(of: maxVal!)
        sudokuArray[row][col] = maxIdx!
    } else {
        sudokuArray[row][col] = 0
    }
} else {
    sudokuArray[row][col] = 0
}
}

let predEndTime = NSDate()
let predTime = predEndTime.timeIntervalSince(predStartTime as Date)

// sudoku 풀이
let solveStartTime = NSDate()
var solvedSudokuArray = sudokuArray
_ = sudoku_solver(&solvedSudokuArray, 0, 0);
let solveEndTime = NSDate()
let solveTime = solveEndTime.timeIntervalSince(solveStartTime as Date)

// 풀어진 sudoku 표시
drawSudoku(solvedSudokuArray, sudokuArray, sudokuImage)
let label = String(format: "Pred: %.3f sec | Solve: %02.3f sec", predTime, solveTime)
infoLabel.text = label
}
}

```

### 코드 15. 수도쿠 풀이 함수

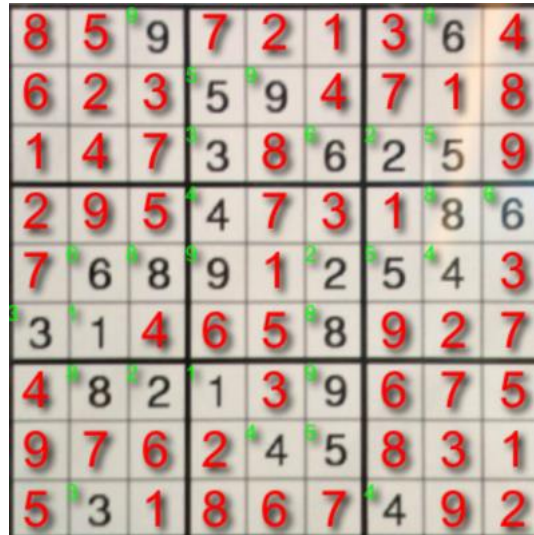
수도쿠 풀이 함수가 불리면 제일 먼저 하는 일이 수도쿠 영역 이미지로부터 9x9로 쪼갠 81개의 이미지를 만들어낸다(OpenCV가 사용됐는데 iPhone에서 OpenCV를 사용하는 방법에 대해서는 다음에 설명한다). 81개의 이미지는 단순히 가로, 세로를 균등하게 9등분한 이미지이다.

다음으로 진행하는 것은 얻어진 81개의 이미지에서 숫자 이미지를 얻는다. 숫자 이미지가 존재하면 return 값 중 boolean으로 true가 반환되어 이를 확인해 CoreML Model을 사용해 숫자 인식을 진행한다. 인식된 숫자를 수도쿠 배열에 기록하고 숫자 이미지가 없으면 수도쿠 배열에 빈 셀로 기록해 놓는다(셀의 값을 0으로 채워 놓는다).

81개의 이미지를 사용해 숫자 인식 후 수도쿠 배열이 만들어지면 수도쿠 풀이를 진행한다. 숫자 인식 단계에서 엉뚱한 숫자가 인식되면 수도쿠 풀이도 제대로 안되므로 인식 단계를 잘 만드는 것이 굉장히 중요하다.

#### 5.4.1. 풀어진 수도쿠 화면에 표시

풀이가 끝난 수도쿠는 화면에 표시하기 위해 drawSudoku() 함수를 사용한다.



8	5	9	7	2	1	3	6	4
6	2	3	5	9	4	7	1	8
1	4	7	3	8	6	2	5	9
2	9	5	4	7	3	1	8	6
7	6	8	9	1	2	5	4	3
3	1	4	6	5	8	9	2	7
4	8	2	1	3	9	6	7	5
9	7	6	2	4	5	8	3	1
5	3	1	8	6	7	4	9	2

그림 17. 풀이가 끝난 수도쿠

그림 17과 같이 인식된 숫자는 해당 셀의 좌상단 귀퉁이에 초록색으로 작게 숫자를 표시하고 풀이로 얻어진 숫자는 해당 셀에 빨간색으로 크게 표시를 했다.

수도쿠 이미지에 있는 숫자와 초록색의 인식된 숫자를 대조해보면 잘 인식된 것을 알 수 있다. 그러나 인식이 잘 못되는 경우는 숫자가 없는 빈공간에 뭔가 숫자가 있는 것으로 인식되거나 혹은 숫자가 잘 못 인식되어 다른 값인 경우가 발생한다. 이런 에러를 줄이는 것이 앞으로 해야할 일이다.

#### 5.5. iPhone에서 OpenCV 사용하기

Python에서는 pip로 설치하고 import 후에 사용하면 되지만 최근의 iPhone은 swift를 기반으로 코딩 되고 OpenCV framework은 C, C++로 제작되기 때문에 직접 사용은 불가능하다.

대신 Objective C++을 기반으로 하는 wrapper를 씌워 이 wrapper에서 OpenCV를 호출하게 하고 swift code에서는 Objective C++로 작성된 wrapper의 함수를 부르도록 하는 방식을 사용한다.

##### 5.5.1. 준비

OpenCV의 iOS Pack은 [www.opencv.org](http://www.opencv.org)에서 받으면 된다. 최신 버전의 다운로드 링크로 들어가면 iOS용 Pack을 받을 수 있다.

XCode에서 project 밑에 Frameworks 이라는 Group을 하나 생성한다(Folder도 같이 생성한다). 그리고 다운로드 받은 opencv2.framework을 복사해 새로 생성된 Frameworks에 넣어 준다.

이어 Finder에서 복사한 opencv2.framework을 drag해서 Frameworks Group 밑에 추가 해주거나 Frameworks Group을 선택하고 'Add Files to ...' 메뉴를 사용해 opencv2.framework을 추가 해준다.

추가한 후 프로젝트 속성의 'Linked Frameworks and Libraries' 항목을 보면 opencv2.framework이 추가 된 것을 확인 할 수 있다.

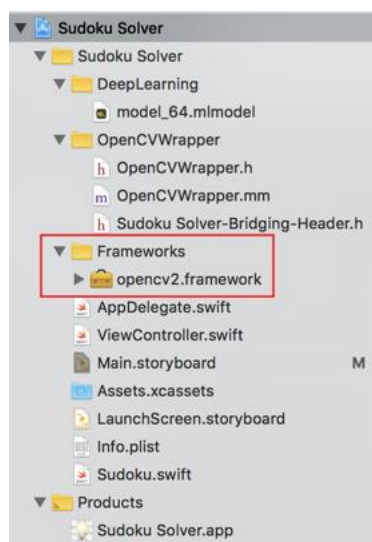


그림 18. opencv2.framework 추가

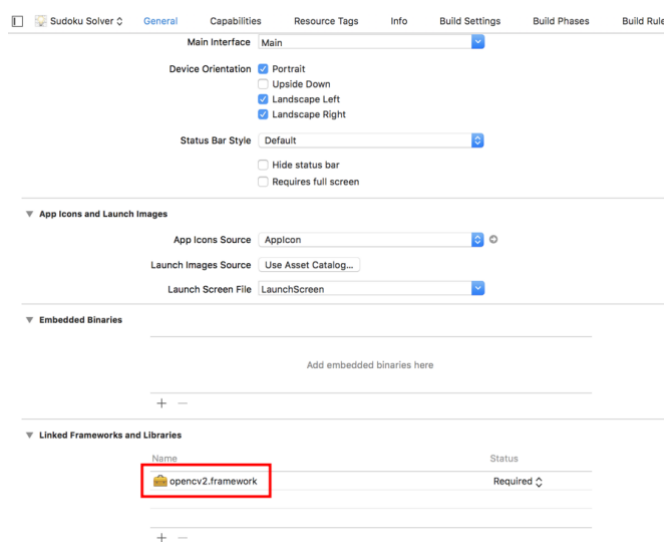


그림 19. 프로젝트 속성 확인

### 5.5.2. OpenCV Wrapper 등록

OpenCV Wrapper를 작성하기 위해서는 'OpenCVWrapper'라는 Group를 프로젝트에 만들고 OpenCVWrapper Group에 'New File' 메뉴를 사용해 새로운 파일을 생성 해준다.

생성할 파일은 우선 OpenCVWrapper.h 이다. 파일 생성 화면에서 파일 타입을 Header File로 선택해 생성한다.

그리고 이어 OpenCVWrapper.mm을 생성한다. 파일 타입은 Objective-C File로 선택한다. 파일 이름은 OpenCVWrapper.mm 으로 하고 Empty File 타입을 선택한다. 파일 위치 선택하면 Bridging Header를 만들 것인지 물어보는데 **반드시** 'Create Bridging Header'를 선택해 브리지 헤더를 생성 해준다.

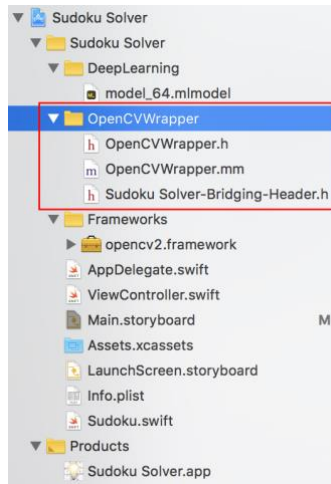


그림 20. OpenCVWrapper 생성

### 5.5.3. Wrapper 수정

```
#import <Foundation/Foundation.h>
#import <UIKit/UIKit.h>

@interface OpenCVWrapper : NSObject

@end
```

코드 16. OpenCVWrapper.h

OpenCVWrapper.h를 열고 코드 16과 같이 수정한다. swift 코드에서는 OpenCVWrapper라는 class로 보이게 된다.

```
#include "OpenCVWrapper.h"
```

코드 17. Bridging-Header.h

Bridging-Header.h를 열고 코드 17과 같이 수정해 준다. Bridging-Header.h에 OpenCVWrapper.h를 넣어주지 않으면 swift 코드에서 OpenCVWrapper가 전혀 보이지 않는다.

```
#import "OpenCVWrapper.h"

#ifdef __cplusplus
#undef NO
#undef YES
#import <opencv2/opencv.hpp>
#import <opencv2/imgcodecs/ios.h>
#endif
```

```
@implementation OpenCVWrapper

@end
```

#### 코드 18. OpenCVWrapper.mm

OpenCVWrapper.mm을 열어 코드 18과 같이 수정해준다. #ifdef 부분은 처음에는 없어도 되지만 OpenCV를 사용하면 에러가 발생해 내가 추가한 것이다.

여기까지 하면 swift 코드에서 OpenCVWrapper.xxx()와 같은 식으로 함수를 호출할 수 있게 된다.

#### 5.5.4. OpenCV 코드 작성

```
#import <Foundation/Foundation.h>
#import <UIKit/UIKit.h>

@interface OpenCVWrapper : NSObject

+ (NSMutableArray *) getSudokuImage: (UIImage *)sourceImage;
+ (NSMutableArray *) getSlicedSudokuNumImages: (UIImage *)sourceImage imageSize: (int)imageSize cutOffset: (int)cutOffset;
+ (NSMutableArray *) getNumImage: (UIImage *)sourceImage imageSize: (int)imageSize;

@end
```

#### 코드 19. OpenCVWrapper.h

OpenCVWrapper.h의 @interface ~ @end 사이에 필요한 함수의 prototype을 선언해준다. operator 등이나 함수 파라미터 선언 방법 등에 대해서는 Objective-C 문서를 참조하기 바란다.

함수 선언 앞의 +표시는 이 함수가 instance method가 아님을 나타내는데 OpenCVWrapper class가 instance로 만들어지지 않아도 불릴 수 있다는 의미이다. -로 선언되면 반드시 호출 전에 class instance가 만들어져야 한다. 위의 코드는 OpenCVWrapper라는 이름에서 의미하듯이 그냥 껍데기에 불과 하기 때문에 instance를 만들어야만 하는 코드가 존재하지 않아 모두 +를 사용한 것이다.

함수들의 return 형식이 NSMutableArray \*인데 Objective-C에서는 함수가 하나의 값 밖에는 return 하지 못한다. 그런데 함수 내부에서 만들어지는 여러 정보를 return 하고 싶어 여러 가지 형식을 넣을 수 있는 array로 만들어 하나만 return하게 만든 것이다.

getSudokuImage()는 입력된 이미지에서 수도쿠 영역을 찾는 함수이고 getSlicedSudokuImages()

는 수도쿠 영역 이미지를 9x9 크기로 잘라내는 함수이다. getNumImage()는 잘라진 이미지에서 숫자 이미지가 있는지 파악해 있으면 숫자 이미지 만을 잘라내 숫자 인식을 위한 입력 이미지로 만들어 주는 함수이다.

```
#import "OpenCVWrapper.h"

#ifdef __cplusplus
#undef NO
#undef YES
#import <opencv2/opencv.hpp>
#import <opencv2/imgcodecs/ios.h>
#endif

#import <cmath>

@implementation OpenCVWrapper

// swift source code에서 cv namespace나 std namespace를 직접 보지 않도록 하기 위해
// std::vector<cv::Point>를 NSArray로 변환해 보낸다.
NSArray *getArrayOfPoint(std::vector<cv::Point> vect) {
    NSMutableArray *resultArray = [[NSMutableArray alloc] init];
    for (int i = 0; i < vect.size(); i++) {
        NSValue *val = [NSValue valueWithCGPoint:CGPointMake(vect[i].x, vect[i].y)];
        [resultArray addObject:val];
    }
    return resultArray;
}

+ (NSMutableArray *) getSukokuImage: (UIImage *)sourceImage {
    @try {
        // UIImage를 cv::Mat로 변환
        cv::Mat sourceMat;
        UIImageToMat(sourceImage, sourceMat);

        //-----
        // 입력 이미지로부터 sudoku rect 찾기
        //-----

        // gray
        cv::Mat grayMat;
        cv::cvtColor(sourceMat, grayMat, CV_BGR2GRAY);

        // // histogram equalize
```

```

// cv::Mat eqHistoMat;
// cv::equalizeHist(grayMat, eqHistoMat);

// threshold
cv::Mat threshMat;
cv::adaptiveThreshold(grayMat, threshMat, 255, CV_ADAPTIVE_THRESH_MEAN_C, CV_THRESH_BINARY_INV, 31,
31);

// find contours
std::vector<std::vector<cv::Point>> contours;
std::vector<cv::Vec4i> hierarchy;
cv::findContours(threshMat, contours, hierarchy, CV_RETR_CCOMP, CV_CHAIN_APPROX_SIMPLE);
if (contours.size() < 1) {
    // no contours found
    return nil;
}

// find max contour
double maxArea = 0;
int maxContourIndex = 0;
for (int i = 0; i < contours.size(); i++) {
    double area = cv::contourArea(contours[i]);
    if (area > maxArea) {
        maxArea = area;
        maxContourIndex = i;
    }
}
std::vector<cv::Point> maxContour = contours[maxContourIndex];

// find rect points from max contour
std::vector<int> sumv, diffv;
for (int i = 0; i < maxContour.size(); i++) {
    cv::Point p = maxContour[i];
    sumv.push_back(p.x + p.y);
    diffv.push_back(p.x - p.y);
}
auto mins = std::distance(std::begin(sumv), std::min_element(std::begin(sumv), std::end(sumv)));
auto maxs = std::distance(std::begin(sumv), std::max_element(std::begin(sumv), std::end(sumv)));
auto mind = std::distance(std::begin(diffv), std::min_element(std::begin(diffv), std::end(diffv)));
auto maxd = std::distance(std::begin(diffv), std::max_element(std::begin(diffv), std::end(diffv)));
std::vector<cv::Point> maxRect;
maxRect.push_back(maxContour[mins]); // top left
maxRect.push_back(maxContour[mind]); // top right

```



```

maxRect.push_back(maxContour[maxs]); // bottom right
maxRect.push_back(maxContour[maxd]); // bottom left

// get width, height of transformed image
cv::Point tl = maxRect[0];
cv::Point tr = maxRect[1];
cv::Point br = maxRect[2];
cv::Point bl = maxRect[3];
double widthA = sqrt(pow((br.x - bl.x), 2) + pow((br.y - bl.y), 2));
double widthB = sqrt(pow((tr.x - tl.x), 2) + pow((tr.y - tl.y), 2));
double heightA = sqrt(pow((tr.x - br.x), 2) + pow((tr.y - br.y), 2));
double heightB = sqrt(pow((tl.x - bl.x), 2) + pow((tl.y - bl.y), 2));
double maxWidth = fmax(int(widthA), int(widthB));
double maxHeight = fmax(int(heightA), int(heightB));

// get transformation matrix & transformed image
cv::Point2f dst[4] = {
    cv::Point2f(0, 0),
    cv::Point2f(maxWidth, 0),
    cv::Point2f(maxWidth, maxHeight),
    cv::Point2f(0, maxHeight)
};

cv::Point2f src[4] = { tl, bl, br, tr };
cv::Mat M = cv::getPerspectiveTransform(src, dst);
cv::Mat warpMat;
cv::warpPerspective(sourceMat, warpMat, M, cv::Size(maxWidth, maxHeight));

NSMutableArray *result = [[NSMutableArray alloc] init];
[result addObject:getArrayOfPoint(maxRect)];
[result addObject:MatToUIImage(warpMat)];

grayMat.release();
threshMat.release();
sourceMat.release();
warpMat.release();

return result;
}

@catch (...) {
    return nil;
}
}

```

```

+ (NSMutableArray *) getSlicedSudokuNumImages: (UIImage *)sourceImage imageSize: (int)imageSize cutOffset:
(int)cutOffset {
    //-----
    // sudoku rect로부터 9x9 slice 만들기
    //-----

    // UIImage를 cv::Mat로 변환
    cv::Mat sourceMat;
    UIImageToMat(sourceImage, sourceMat);

    std::vector<UIImage*> slicedImages;
    cv::Mat numImageMat = cv::Mat(imageSize * 9, imageSize * 9, CV_8UC4);

    double dx = (sourceMat.size()).width / 9.0;
    double dy = (sourceMat.size()).height / 9.0;
    if (dx < 1.0 || dy < 1.0) {
        // 입력 이미지가 너무 작은 경우는 처리하지 않음
        return nil;
    }
    for (int row = 0; row < 9; row++) {
        for (int col = 0; col < 9; col++) {
            int x = (int)(col * dx);
            int y = (int)(row * dy);
            cv::Rect r = cv::Rect(x + cutOffset, y + cutOffset, dx - cutOffset, dy - cutOffset);
            cv::Mat sliced = sourceMat(r);
            cv::Mat resized;
            cv::resize(sliced, resized, cv::Size(imageSize, imageSize));

            slicedImages.push_back(MatToUIImage(resized));

            // slice 된 것들을 모아 하나의 이미지로 만든다
            x = col * imageSize;
            y = row * imageSize;
            r = cv::Rect(x, y, imageSize, imageSize);
            resized.copyTo(numImageMat(r));

            resized.release();
        }
    }

    // sliced image를 NSArray로 변경
    NSArray *numImages = [NSArray arrayWithObjects:&slicedImages[0] count:slicedImages.size()];
    // return numImages;
}

```

```

// 여러 가지 정보를 묶어 return
NSMutableArray *result = [[NSMutableArray alloc] init];
[result addObject:numImages];
[result addObject:MatToUIImage(numImageMat)];

sourceMat.release();
numImageMat.release();

return result;
}

+ (NSMutableArray *) getNumImage: (UIImage *)sourceImage imageSize: (int)imageSize {
    // UIImage를 cv::Mat로 변환
    cv::Mat sourceMat;
    UIImageToMat(sourceImage, sourceMat);

    // gray
    cv::Mat grayMat;
    cv::cvtColor(sourceMat, grayMat, CV_BGR2GRAY);

    // threshold
    cv::Mat threshMat;
    cv::adaptiveThreshold(grayMat, threshMat, 255, CV_ADAPTIVE_THRESH_MEAN_C, CV_THRESH_BINARY_INV, 11, 41);

    // find contours
    std::vector<std::vector<cv::Point>> contours;
    std::vector<cv::Vec4i> hierarchy;
    cv::findContours(threshMat, contours, hierarchy, CV_RETR_CCOMP, CV_CHAIN_APPROX_SIMPLE);
    if (contours.size() < 1) {
        // no contours found
        return nil;
    }

    cv::Mat contourMat;
    cv::cvtColor(threshMat, contourMat, cv::COLOR_GRAY2RGB);

    int gx = (int)(sourceMat.size().width * 0.05);
    int gy = (int)(sourceMat.size().height * 0.05);
    int gw = (int)(sourceMat.size().width * 0.9);
    int gh = (int)(sourceMat.size().height * 0.9);
    cv::rectangle(contourMat, cv::Point(gx, gy), cv::Point(gx+gw, gy+gh), CV_RGB(0, 255, 0), 2);

```

```

// find max contour
int maxArea = 0;
int maxContourIndex = -1;
cv::Rect bRect;
for (int i = 0; i < contours.size(); i++) {
    double area = cv::contourArea(contours[i]);
    if (area > 10) {
        cv::Rect r = cv::boundingRect(contours[i]);
        // overlap 영역 찾기
        int ox = MAX(gx, r.x);
        int oy = MAX(gy, r.y);
        int ox2 = MIN(gx+gw, r.x+r.width);
        int oy2 = MIN(gy+gh, r.y+r.height);
        int ow = ox2 - ox;
        int oh = oy2 - oy;
        int oarea = ow * oh;
        // overlap 영역의 크기가 bounding rect와 같으면 bounding rect를 포함하고 있는 것
        if (oarea == r.area()) {
            // 포함되는 것 중 제일 큰 것만 남긴다
            if (area > maxArea) {
                maxArea = area;
                maxContourIndex = i;
                bRect = r;
            }
        }
    }
}

// 여러 가지 정보를 묶어 return
NSMutableArray *result = [[NSMutableArray alloc] init];

// maxContourIndex가 0이상이면 숫자가 있는 것
if (maxContourIndex >= 0) {
    cv::drawContours(contourMat, contours, maxContourIndex, CV_RGB(0, 255, 0), 2);
    NSNumber *t = [NSNumber numberWithInt:true]; // true를 return
    [result addObject:t];
} else {
    NSNumber *f = [NSNumber numberWithInt:false]; // false를 return
    [result addObject:f];
}

// 디버깅을 위한 정보 제공 목적
// contourMat는 95% box와 안의 숫자 contour가 그려진 이미지

```

```

[result addObject:MatToUIImage(contourMat)];

grayMat.release();
threshMat.release();
contourMat.release();

return result;
}

@end

```

### 코드 20. OpenCVWrapper.mm

코드 20은 코드 1에서와 같이 Python으로 구현한 테스트 코드를 Objective-C++로 변경한 것이다. Python의 Numpy나 Array 다루는 방법 등이 C++에는 없기 때문에 그리고 Python에서 보이는 OpenCV가 C++에서 보이는 형식과 많이 다르기 때문에 그에 맞춰 코딩을 다시 했다.

그러나 알고리즘은 다르지 않고 그대로 사용했고 단지 일부 값은 실제 iPhone에서 실행해 가면서 튜닝을 했다.

중요한 것은 alloc 된 메모리나 object에 대해서는 사용이 끝나면 없애 주는 것이 좋다. 그렇지 않으면 실행을 반복함에 따라 메모리 사용량이 점점 증가해 나중에는 crash가 발생하기도 한다. 실제 측정에서는 큰 차이는 없었지만 그래도 항상 필요 없는 object에 대해서 없애주는 버릇을 들여 놓는 것이 버그를 줄이는 지름길이라 생각된다.

## 6. 결과

구현된 앱은 그림 21과 같다. 문자 인식에 약 0.4초, 수도쿠 풀이에 약 0.12초가 걸렸다. 실제 동작을 시켜보면 상당히 빨리 문자 인식이 진행되는 것을 알 수 있다.



그림 21. 최종 동작 화면

## 7. 할 일

해결 해야할 일은 2가지 정도로 생각하고 있고 언젠가는 업데이트 예정이다.

- ① 숫자 인식을 잘라낸 이미지로 바로 진행하게 하는 것
- ② 수도쿠 풀이 중에 무한 루프에 빠지는 경우가 있는데 해결 혹은 취소 할 방법 찾기

### 7.1. 숫자 인식 개선

현재 CoreML Model은 10개의 숫자 classification을 진행하는데 예를 들어 이미지는 1을 나타내나 인식된 숫자는 7과 같은 경우 분명히 prediction의 확률 값이 낮음에도 불구하고 그 중에 그나마 큰 값이 7에 해당하는 것이라 7로 인식이 됐는데 확률 값이 일정 기준보다 낮거나 아예 숫자가 없는 이미지의 경우 -1과 같은 결과를 내도록 10개의 class를 11개로 확장할 필요가 있다고 생각이 들었다.

이래야 입력 이미지에서 숫자가 있는지 판단할 필요 없이 잘라낸 것을 그냥 인식하도록 만들면

되니 좀더 편하고 빠른 구조가 될 것이다.

## 7.2. 무한 루프 해결

숫자 인식이 잘 못된 경우 수도쿠 풀이를 하면 가끔 무한 루프에 빠져버리는 경우가 발생한다. 지금은 앱을 죽였다가 다시 실행하고 있는데 적어도 손가락으로 터치를 하면 수도쿠 풀이를 멈추고 카메라로 사진을 찍을 수 있도록 돌아가게는 만들 필요가 있어 보인다.

문제는 recursive하게 돌고 있는 풀이 알고리즘이 스스로 잘 못 됐다는 것을 어떻게 알 것인가?가 문젠데 일단은 외부에서 컨트롤 할 수 있는 flag를 두고 진행 중에 flag가 세팅되면 풀이를 하지 않고 멈추게 하는 것을 구현할 예정이다.