University of Bucharest

Faculty of Mathematics and Computer Science

Computer Vision

Automatic score calculator for Mathable

Chirobocea Mihail Bogdan

2023-2024

Content:

Purpose and tasks

- ## Purpose

The purpose of this project is to implement an automatic score calculation system for the game Mathable. We are using a set of 200 photos from 4 games of Mathable - 50 photos per game, one photo per move - taken in a slightly controlled environment. The board may vary in position, rotation and angle, as well as the color balance, saturation, luminosity and shadows may slightly vary from one photo to another.

- ## Tasks
    - Task 1:

For each image, the algorithm must provide the piece positions of the letters that were placed on the board in the respective move.

    - Task 2:

For each piece position, the algorithm has to recognize the number that is on that track

    - Task 3:

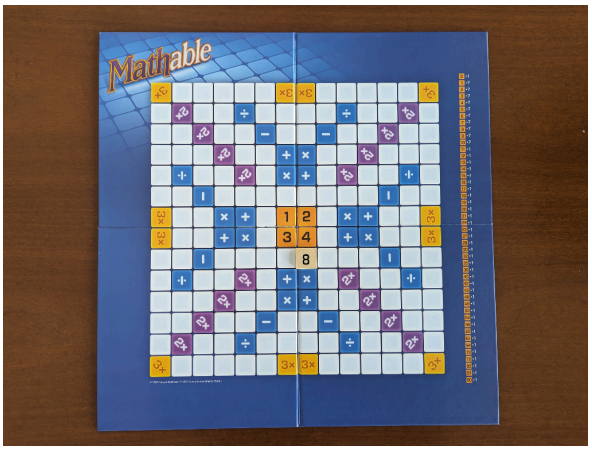
For each turn, the algorithm has to calculate the score obtained by the player in the respective round.

Task 1 - finding tokens position

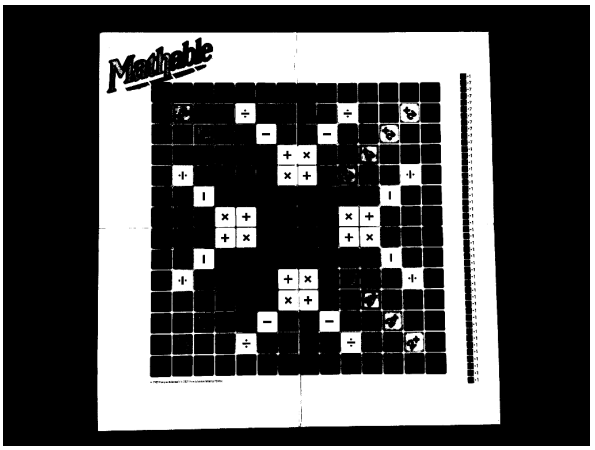
Solving idea

First, we want to cut the board from the image. In order to do that we are going to use HSV (Hue, Saturation, Value) masks.:

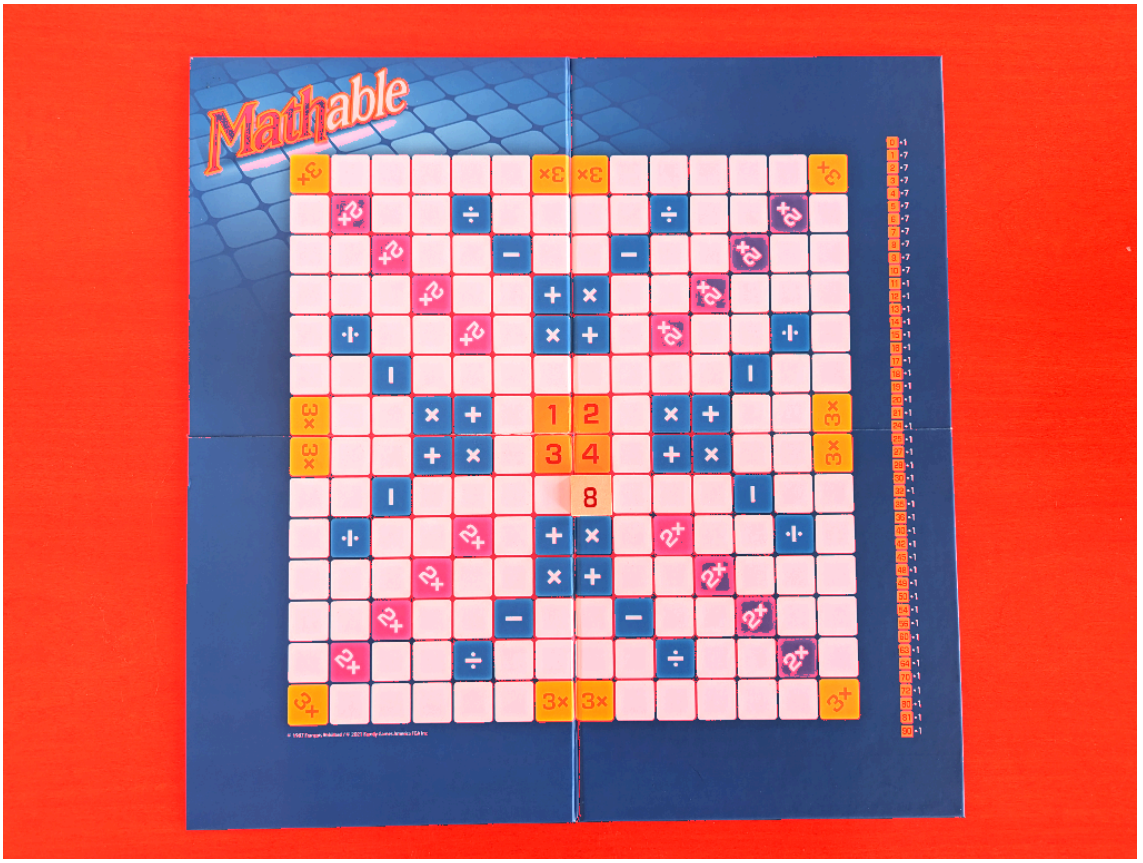
Original image



Blue mask

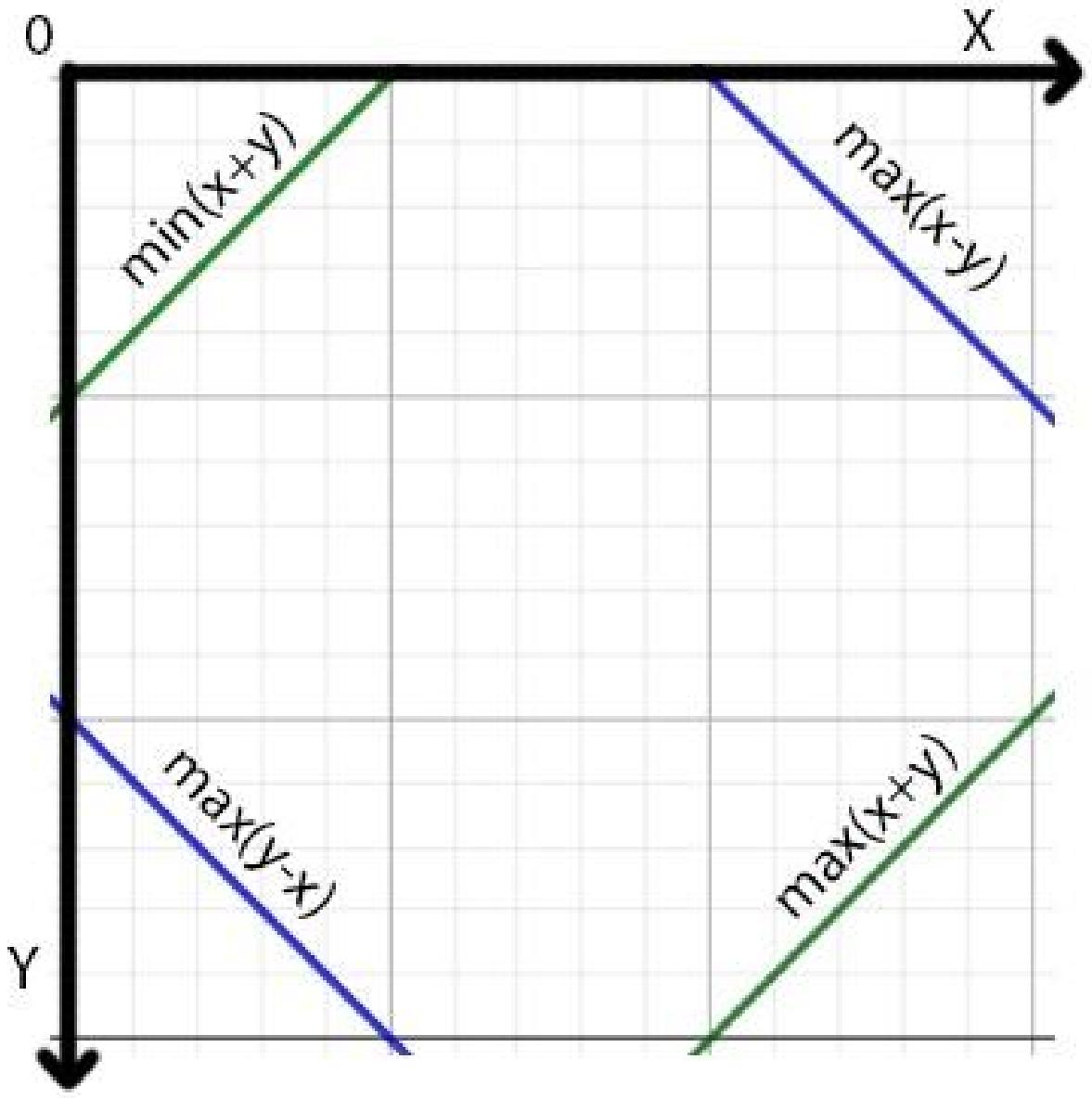


Original image overlaid with the blue mask (represented with red)



Now, we want to find the corners of the board. We want to find the white dots with coordinates (x,y) which have:

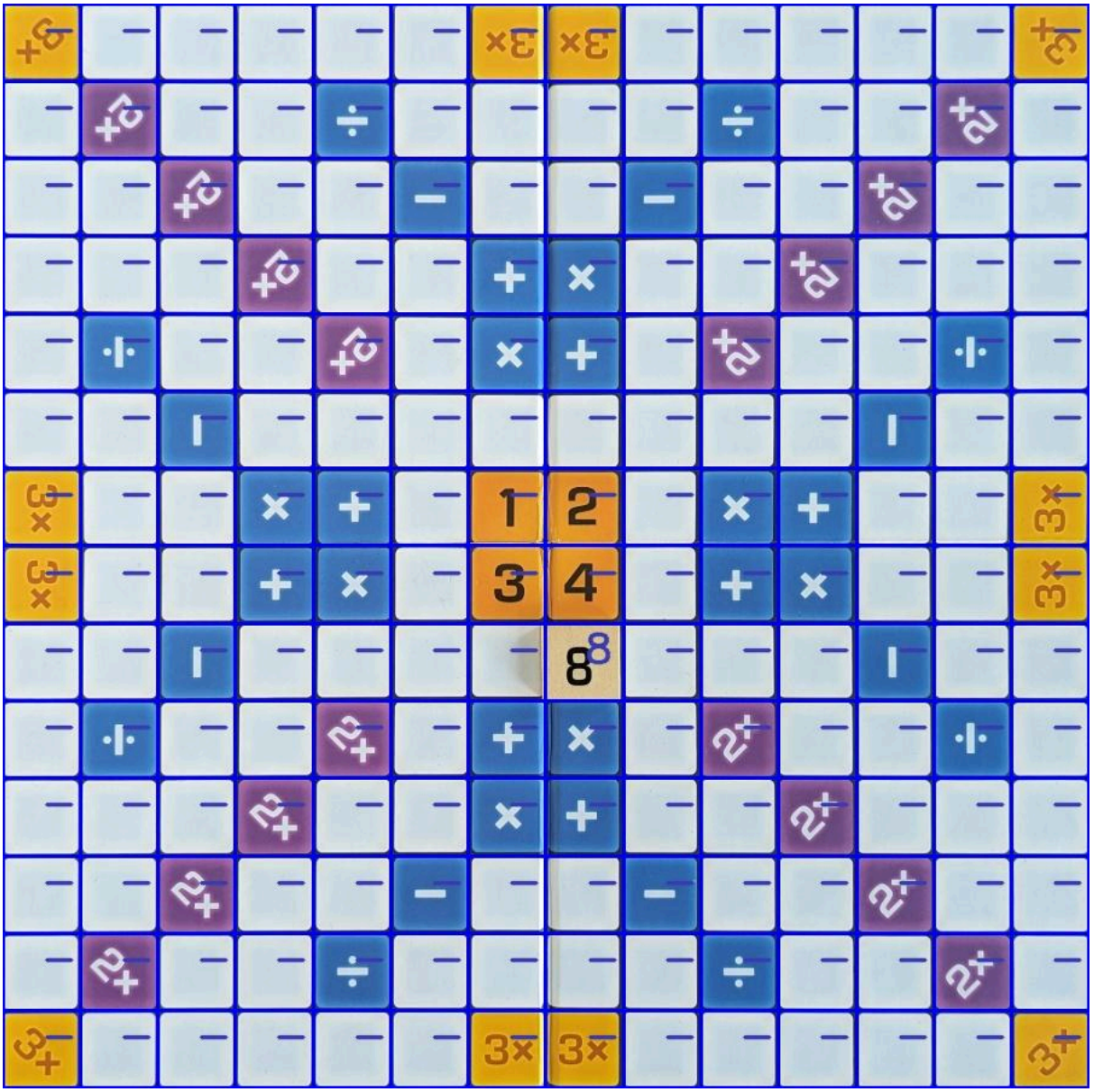
- The minimum x+y value, for top left corner
- The maximum x-y value, for top right corner
- The maximum x+y value, for bottom right corner
- The maximum y-x value, for bottom left corner



Keep in mind that the top left corner in a photo starts with coordinates (0,0).

Now that we have the corners of the board we can make a warp crop to remove distortion of the photo (see cod). The image is also sized to 900x900 inside this warp operation.

We are going to divide the board into 14 equal parts in length and 14 equal parts in width, thus we will obtain the 196 squares of the board game.



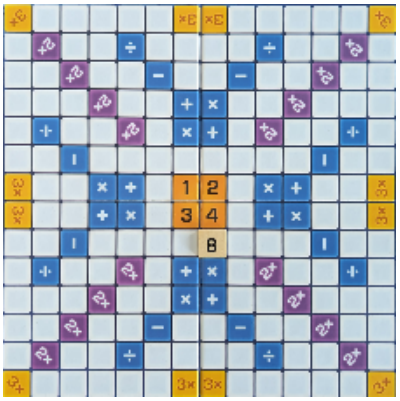
We can make a mask to fit only the background color of the tokens. Based on that mask made on each square we will find out if there is a letter or not.

For this mask, we firstly apply an dilation of kernel 16x16 in order to remove possible noise followed by an erosion operation of kernel 26x26 in order to strengthen back our tokens.

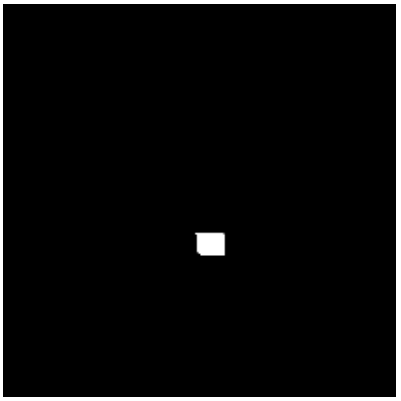
We cut patches with 20 pixels less than the actual square in order to avoid slightly moved tokens and check if the average intensity is above 128. If so, we consider that there is a token, else we consider it's empty.

To find the new tokens, we compare the previous map with the current one.

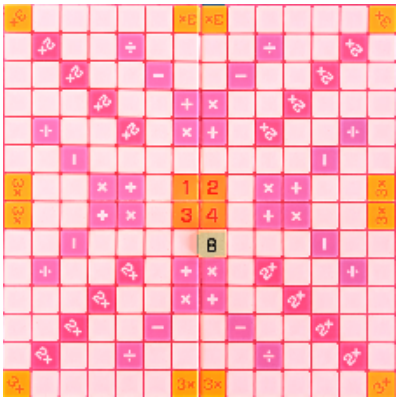
Board



Token mask



Board overlaid with the token mask



How does the code work?

First, we define the HSV masks and apply erosion and dilation as needed.

```
def get_mask(image, lower_bound:tuple[int, int], upper_bound:tuple[int, int], dilate_kernel_size:int = None,
erode_kernel_size:int = None):

    hsv = cv.cvtColor(image, cv.COLOR_BGR2HSV)

    mask = cv.inRange(hsv, lower_bound, upper_bound)
    if dilate_kernel_size is not None:
        mask = cv.dilate(mask, np.ones((dilate_kernel_size, dilate_kernel_size), np.uint8), iterations=1)
    if erode_kernel_size is not None:
        mask = cv.erode(mask, np.ones((erode_kernel_size, erode_kernel_size), np.uint8), iterations=1)

    return mask
```

After that, we need to find the corners of the board (represented by the mask).

```
for point in contours[i].squeeze():
    if possible_top_left is None or point[0] + point[1] < possible_top_left[0] + possible_top_left[1]:
        possible_top_left = point
    if possible_bottom_right is None or point[0] + point[1] > possible_bottom_right[0] + possible_bottom_right[1]:
        possible_bottom_right = point
    if possible_top_right is None or point[0] - point[1] > possible_top_right[0] - possible_top_right[1]:
        possible_top_right = point
    if possible_bottom_left is None or -point[0] + point[1] > -possible_bottom_left[0] + possible_bottom_left[1]:
        possible_bottom_left = point
```

Here we apply the idea of finding x, y which have maximum/minimum values for some functions. [0] represents the x axis and [1] the y axis. contours[i] is a component of connected white pixels in the mask. We have to go through all components to find the maximum area i.e. the area of the border.

```
if cv.contourArea(np.array([[possible_top_left],[possible_top_right],[possible_bottom_right],[possible_bottom_left]])) > max_area:
    max_area = cv.contourArea(np.array([[possible_top_left],[possible_top_right],[possible_bottom_right],[possible_bottom_left]]))
    top_left = possible_top_left
    bottom_right = possible_bottom_right
    top_right = possible_top_right
    bottom_left = possible_bottom_left
```

Now we will use the built-in function from open-cv to wrap the image and cut the board.

```python
def crop_board(top_left,top_right,bottom_right,bottom_left, new_size, image, crop_size):

    puzzle = np.array([top_left,top_right,bottom_right,bottom_left], dtype = "float32")
    destination_of_puzzle = np.array([[0,0],[new_size,0],[new_size,new_size],[0,new_size]], dtype = "float32")
    M = cv.getPerspectiveTransform(puzzle, destination_of_puzzle)

    board = cv.warpPerspective(image, M, (new_size, new_size), flags = cv.BORDER_REFLECT + cv.WARP_FILL_OUTLIERS)

    y, x, _ = board.shape
    start_x = x//2 - crop_size//2
    start_y = y//2 - crop_size//2

    return board[start_y:start_y+crop_size, start_x:start_x+crop_size]
```

We apply again an HSV mask using the above function, and iterate through patches in order to calculate their mean. We note with “+” the positions in the matrix where we have a letter and with “-” for empty positions.

```python
def board_configuration(board, lines_horizontal, lines_vertical):

    matrix = [[0 for i in range(len(lines_vertical)-1)] for j in range(len(lines_horizontal)-1)]
    for i in range(len(lines_horizontal)-1):
        for j in range(len(lines_vertical)-1):

            patch = cut_patch(board, lines_horizontal, lines_vertical, i, j, 20, 20)
            if np.mean(patch) > 128:
                matrix[i][j] = '+'
            else:
                matrix[i][j] = '-'

    return matrix
```

To find only the new changes, we iterate through the current and previous versions and check for discrepancies.

```python
def find_new_changes(last_matrix, current_matrix):

    if last_matrix is None:
        return current_matrix.copy()
    else:
        diff = np.empty_like(last_matrix)
        for i in range(diff.shape[0]):
            for j in range(diff.shape[1]):
                if last_matrix[i][j] == '-' and current_matrix[i][j] != '-':
                    diff[i][j] = current_matrix[i][j]
                else:
                    diff[i][j] = '-'
        return diff
```

Task 2 - classifying tokens

Solving idea

Here we will use the concept of template matching, for this we will need masks for each letter. Those masks were made by saving the warped board (from task 1) and after that taking that image into photoshop. There we used a series of adjustments like: denoise, contrast, painting, masking.

We found out that using the smallest crop of each number gives better results in our whole pipeline, even if it's unfair to make scores using different sizes. We made only one exception with the number "1" which is so tight that it easily gets good results in a lot of wrong cases, so we extended the template into the sides with a few pixels.

Using the above approach generates big problems with numbers greater than 9, as into a 10 there will be a got fit of an 1 and a 0 also. To fix this issue we take the greatest 2 scores for numbers from 0 to 9, and if those 2 have a small enough difference between them and each with a big enough score, we consider that number as a composed one. So we check the possible numbers composed from those 2 and keep the one with the highest score. For example, if we found great scores for both 1 and 2 we will check the matching with the numbers 21 and 12, keeping the one with the best score. Else, we simply keep the number from 0 to 9 with the highest score.
There is one more exception with the number 11 which is composed only with the number 1 but twice. In case we have a great score for number 1, we also check the score with number 11 and also the score with the extended (via padding) template of 1 up to the same size as the template of number 11. We keep the one with the highest score.

This whole process also makes use of masking, erosion and dilation in order to remove noise and better fit our templates. We also extend our patch cut or pad it.

How does the code work?

We firstly preprocess the patch and also the template.

```
scale_factor = 2
gray_patch = cv.cvtColor(patch, cv.COLOR_BGR2GRAY)
gray_patch = cv.inRange(gray_patch, ( 0), (90))
show_image(gray_patch, resize_factor=1)
gray_patch = cv.erode(gray_patch, np.ones((3, 3), np.uint8), iterations=1)
gray_patch = cv.dilate(gray_patch, np.ones((6, 6), np.uint8), iterations=1)
gray_patch = cv.resize(gray_patch, (gray_patch.shape[1] * scale_factor, gray_patch.shape[0] * scale_factor), interpolation = cv.INTER_LINEAR)
show_image(gray_patch, resize_factor=1)
```

```
for j in range(1, len(letters)+1):

    img_template=cv.imread(mask_location+'/'+letters[j]+'.jpg')
    img_template=cv.cvtColor(img_template, cv.COLOR_BGR2GRAY)
    img_template = 255 - img_template
    top, bottom, left, right = [1, 1, 1, 1]
    img_template = cv.copyMakeBorder(img_template, top, bottom, left, right, cv.BORDER_CONSTANT, value=0)
    img_template = cv.dilate(img_template , np.ones((3, 3), np.uint8), iterations=1)
    img_template = cv.resize(img_template, (img_template.shape[1] * scale_factor, img_template.shape[0] * scale_factor), interpolation = cv.INTER_LINEAR)
```

After that we take the scores for all templates.

We make a new template for number 1 that has the same size as the template of number 11.

```
if j == 2:
    img_teamplate2 = img_template.copy()
    x_axis = (max_numar[1] - img_teamplate2.shape[1]) // 2
    y_axis = (max_numar[0] - img_teamplate2.shape[0]) // 2
    top, bottom, left, right = [y_axis, y_axis, x_axis, x_axis]
    img_teamplate2 = cv.copyMakeBorder(img_teamplate2, top, bottom, left, right, cv.BORDER_CONSTANT, value=0)
    corr2 = cv.matchTemplate(gray_patch, img_teamplate2, cv.TM_CCOEFF_NORMED)
    corr2 = np.max(corr2)
```

Now we check if our number might be composed or not.

```
cifers = scores[:10]

[c1, c2], [i1, i2] = two_greatest_numbers(cifers)

if (i1 == 1 and c1 > 0.8) or (i2 == 1 and c2 > 0.8):
    # print("Special", scores[11], corr2)
    if scores[11] > 0.6 or corr2 > 0.6:
        if corr2 > scores[11]:
            letter = "1"
        else:
            letter = "11"
    else:
        letter = None

if (abs(c1-c2) < 0.21 and c1 > 0.79 and c2 > 0.79) or (abs(c1-c2) < 0.1 and 0.65 < c1 < 0.9 and 0.65 < c2 < 0.9):
    letter = get_composed_letter(scores, y_values, i1, i2, letters)
```

Task 3 - score calculation

Solving idea

Here the idea is to keep a matrix that contains only the new tokens (added in the current round of the turn) and one matrix with all other tokens. Doing so we will be able to find if a new letter is in a special position or not and it will be easy to find all the new equations formed.

How does the code work?

We check if the new number is in a special position.

```
if (x, y) in plus_constraint:
    plus_c = True
elif (x, y) in minus_constraint:
    minus_c = True
elif (x, y) in multiply_constraint:
    multiply_c = True
elif (x, y) in divide_constraint:
    divide_c = True
elif (x, y) in x3_points:
    bonus = 3
elif (x, y) in x2_points:
    bonus = 2
```

After that we check for equations in each direction.

```
if x >= 2:  # Ensure there are two points above
    if current_matrix[x-1][y] != '-' and current_matrix[x-2][y] != '-':
        if check_equation(current_matrix[x-1][y], current_matrix[x-2][y], diff_matrix[x][y], plus_c, minus_c, multiply_c, divide_c):
            score_power += 1

if x <= 11:  # Ensure there are two points below
    if current_matrix[x+1][y] != '-' and current_matrix[x+2][y] != '-':
        if check_equation(current_matrix[x+1][y], current_matrix[x+2][y], diff_matrix[x][y], plus_c, minus_c, multiply_c, divide_c):
            score_power += 1

if y >= 2:  # Ensure there are two points to the left
    if current_matrix[x][y-1] != '-' and current_matrix[x][y-2] != '-':
        if check_equation(current_matrix[x][y-1], current_matrix[x][y-2], diff_matrix[x][y], plus_c, minus_c, multiply_c, divide_c):
            score_power += 1

if y <= 11:  # Ensure there are two points to the right
    if current_matrix[x][y+1] != '-' and current_matrix[x][y+2] != '-':
        if check_equation(current_matrix[x][y+1], current_matrix[x][y+2], diff_matrix[x][y], plus_c, minus_c, multiply_c, divide_c):
            score_power += 1
```