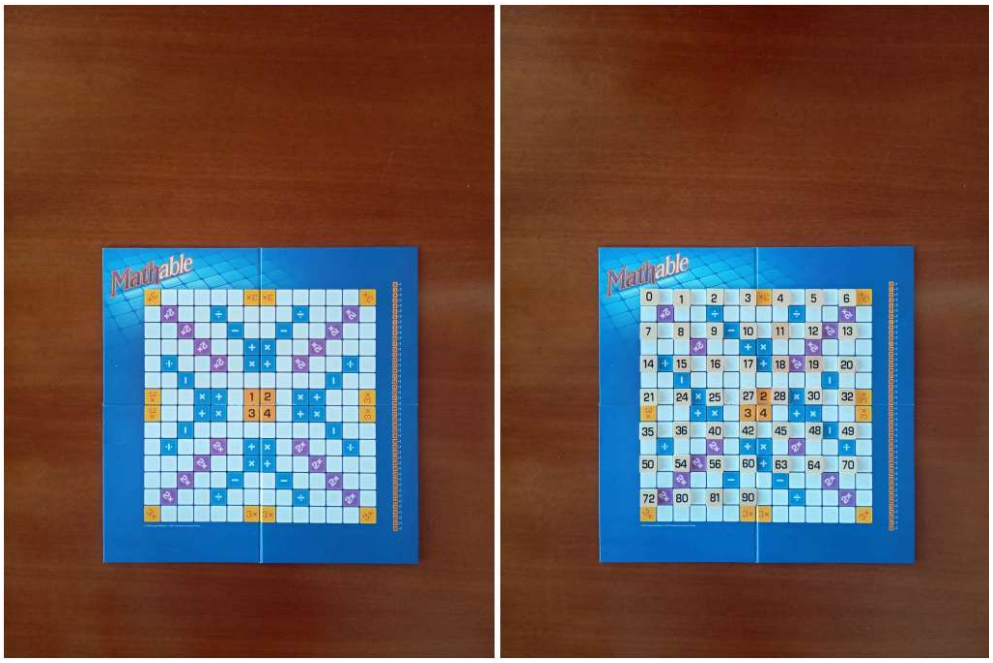
Project Report
Automatic computing of score in the Mathable game

Introduction and Objectives

This project is a homework for the *Computer Vision* class and focuses on applying various techniques to **extract visual information** from rounds of Mathable and compute the score of the given round.



The above picture shows an empty game table (left) and every type of piece placed on the game table (right).

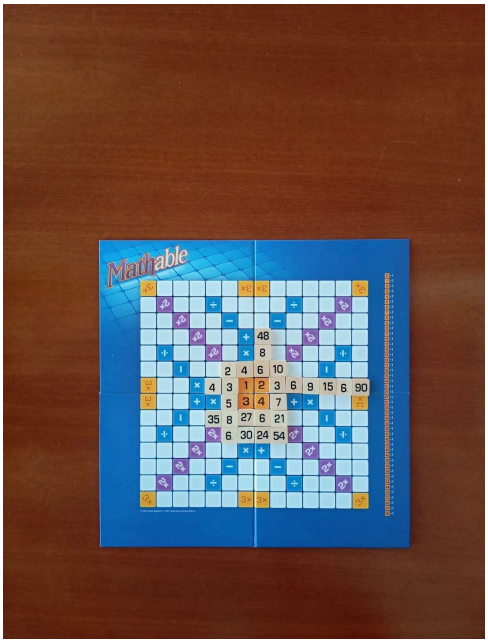
Solving the given problem can be broken down into **3 tasks**:

i. Detecting the position of the last placed piece
ii. Recognizing the number on the last placed piece
iii. Implementing score computation based on the rules

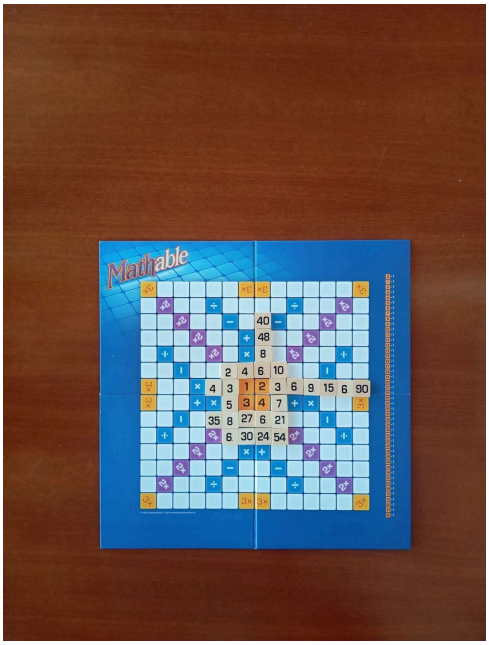
The following pages will cover the implementation details for those 3 tasks.

i. Detecting the position of the last placed piece

This task will be demonstrated on pictures *2_25.jpg* and *2_26.jpg* from the *antrenare* directory.

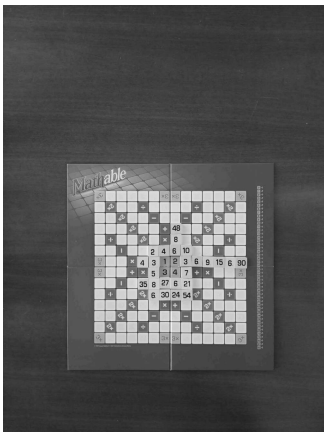


2_25.jpg

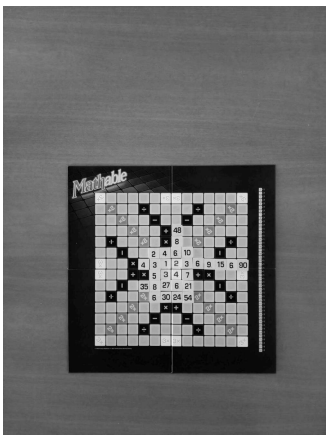


2_26.jpg

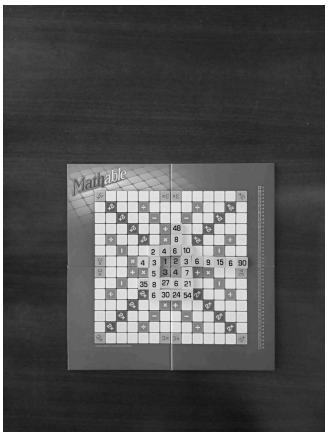
To detect the position of the last placed piece (the *40* from the 3rd row in our example case) we have to **extract the playing area** from our pictures. The following images represent different ways of transforming the image into **grayscale**: OpenCV's COLOR_BGR2GRAY, selecting only the red channel, selecting only the green channel and selecting only the blue channel.



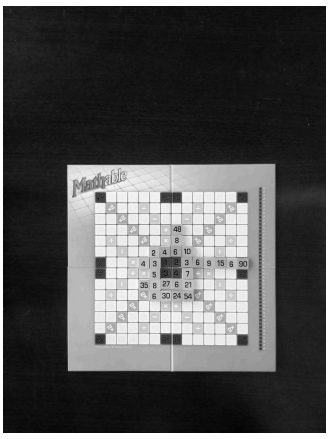
COLOR_BGR2GRAY



Red channel



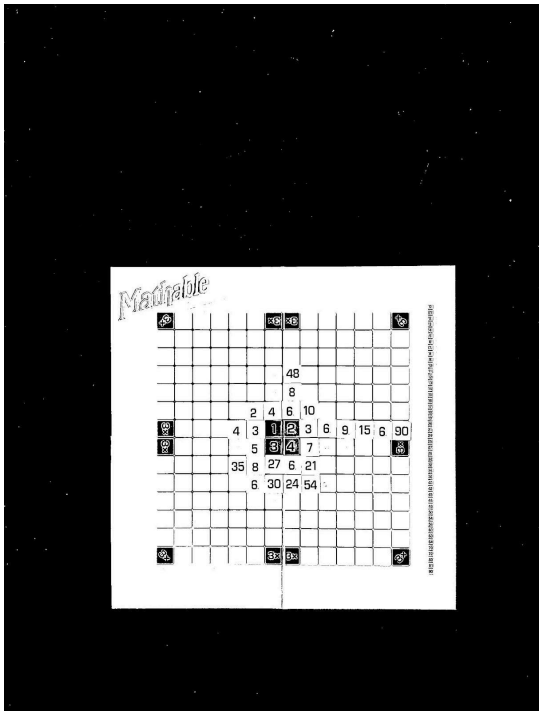
Green channel



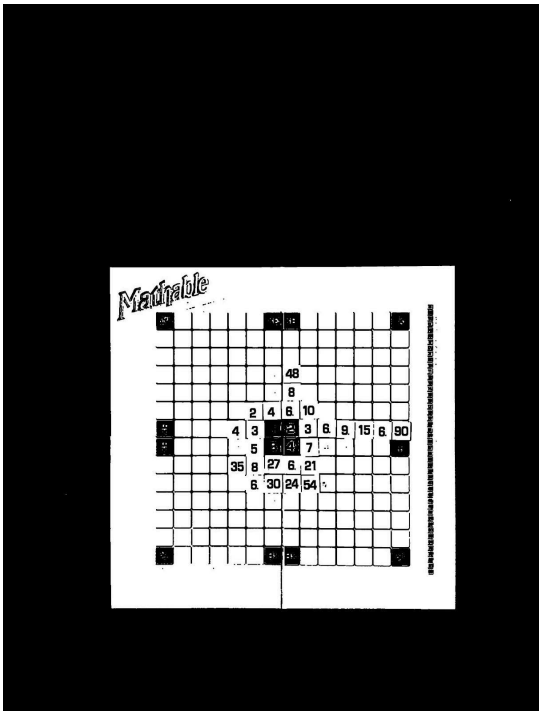
Blue channel

As we can see from the experiment in the pictures above, we get the biggest **intensity difference** between the table and the game board by using the **blue channel** of the original image, which could even be deducted by just looking at the original image. Thus, we will continue using the blue channel picture for this task of the project.

The next step is to apply some **blurring and sharpening filters** to better define the edges of the game board. After this is done, we will apply a **threshold** over the image and use an **erosion operation**.



Threshold



Threshold after erosion

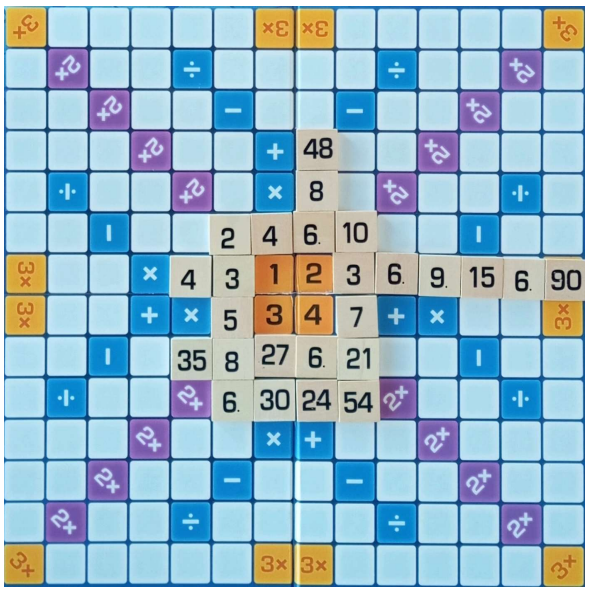
Now we will find all the contours in the picture using OpenCV's *findContours* function. After that, we will **iterate through the contours** with at least 3 points and select the top-left, top-right, bottom-left and bottom-right points of the biggest contour, which is found using the *contourArea* function.

Detected corners on original image

Now we can copy the game board to another array and, using the *getPerspectiveTransform* and *warpPerspective* functions, get the result we wanted.

Resulting game board

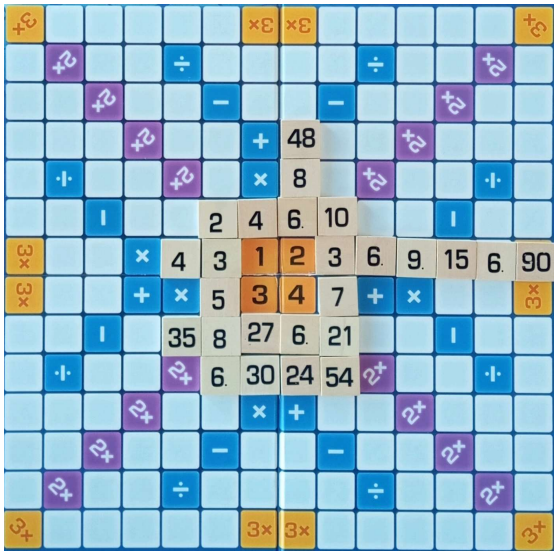
The last processing that needs to be done is **removing the bordering blue** area to get only the part that is relevant for our project: the play area.



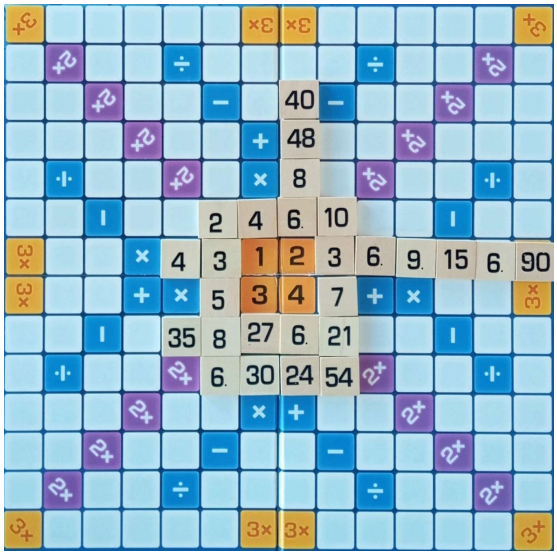
Resulting play area

The code that achieves all the steps above and returns the play area can be found in the *get_play_area* function from the *main.ipynb* notebook.

Now, we can even better observe the position of the last added piece (the *40* from the 3rd row) but we will need to use some more techniques so that algorithm can do it as easily as us.

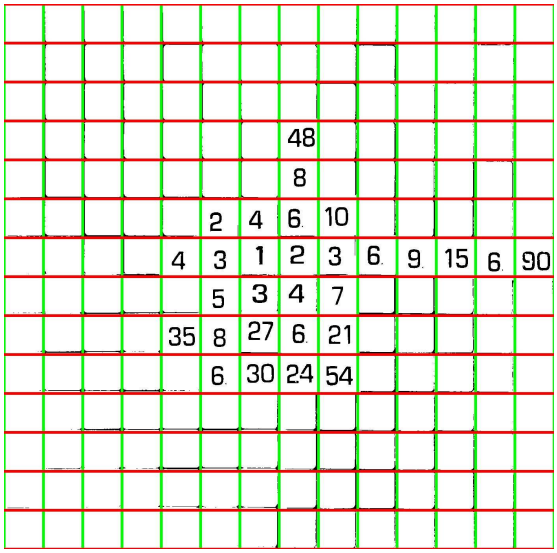


Play area of *2_25.jpg*

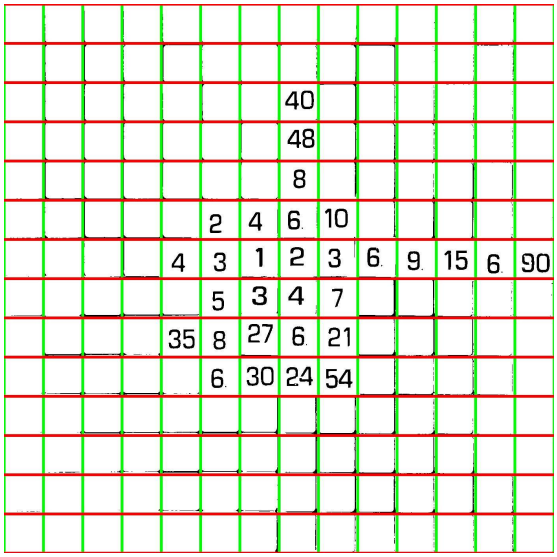


Play area of *2_26.jpg*

First, we will convert the images to **grayscale**, apply a **threshold** on both and divide them in 14x14 squares of equal size.



Thresholded and divided *2_25.jpg**



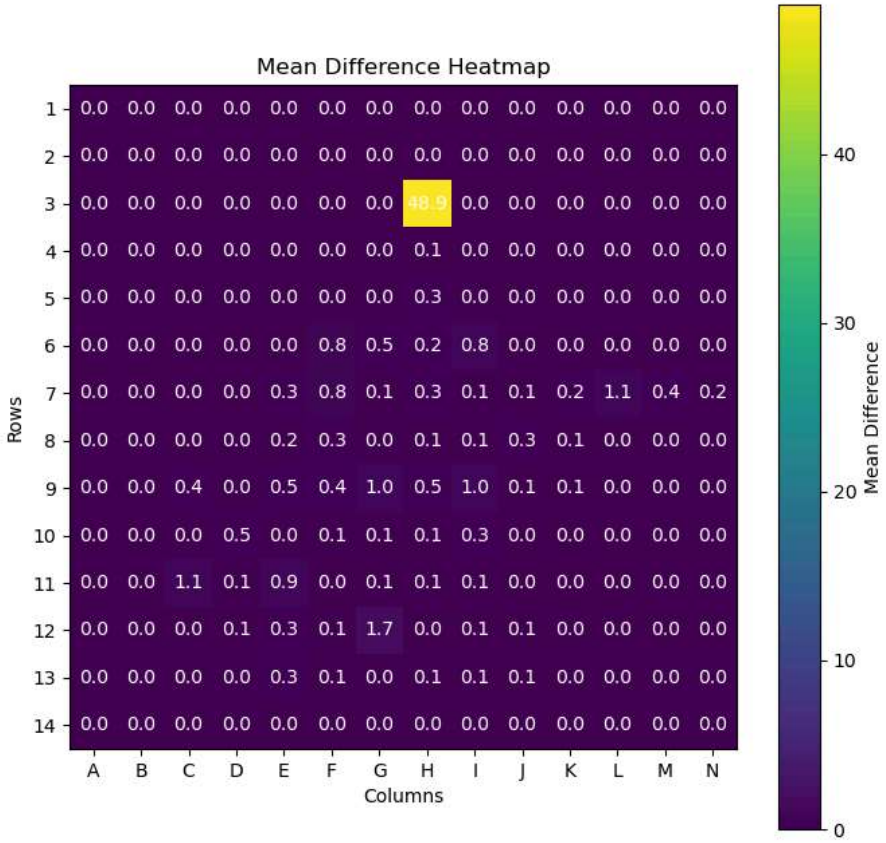
Thresholded and divided *2_26.jpg**

**To be noted that the green and red lines are not added to the pictures that we are working it, they are here just for a better visualisation*

For every square in the 14x14 grid we will compute the following operations:

- Compute the **mean of the values** in the "previous move" picture (*2_25.jpg*)
- Compute the **mean of the values** in the "current move" picture (*2_26.jpg*)
- Compute the **absolute value of the difference** between those values

The **square with the largest value** of the difference of the means is the square where the new piece was placed. For more insight, we can save the values of the differences in a matrix and plot a **heatmap**.



As we can see, a **slight variation** exists in most of the squares, probably because of the natural light conditions in which the photos were taken, however, the variation is not large enough to cause problems with our approach.

This **concludes the first task** of the project. The code for the detection of the last piece can be found in the *get_placed_piece_position* function and the plotting of the heatmap can be found in the *plot_heatmap* function.

ii. Recognizing the number on the last placed piece

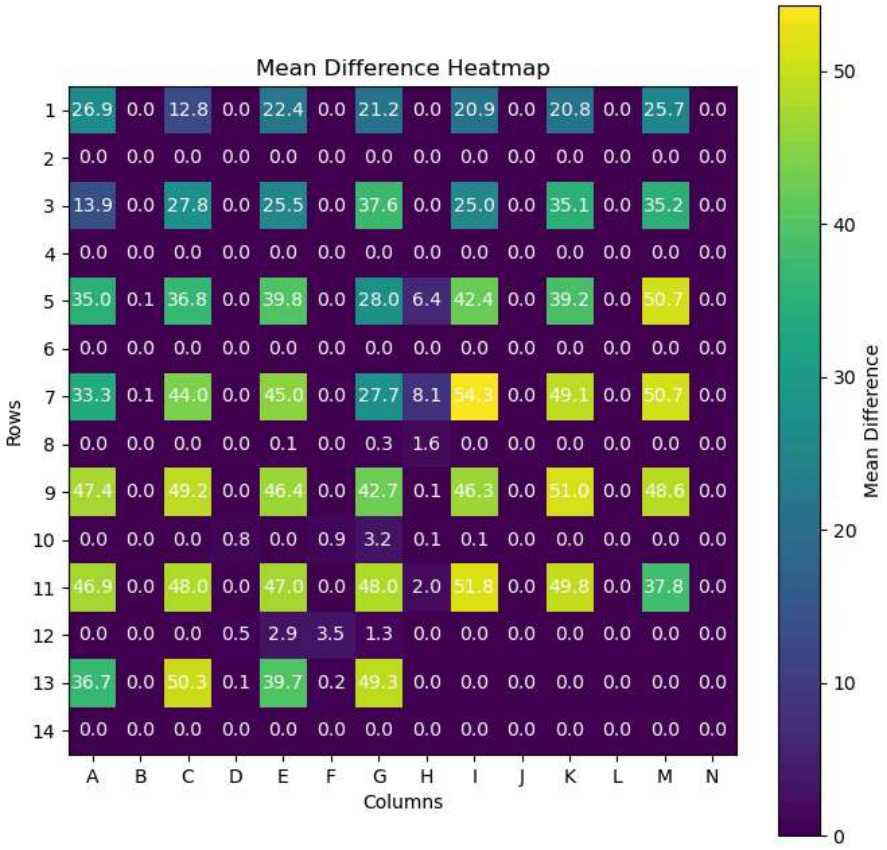
This task will be demonstrated on the new piece added in the *2_26.jpg* picture from the *antrenare* directory.

Piece number to recognize

The above picture is the square where the new piece was placed from *task i*. For classifying it, we will use OpenCV's *matchTemplate* function. But first, we need to **construct our templates**. The code that does this can be found in the *get_piece_patches* function and uses the same techniques as *task i*. The difference is that we save the **46 squares** with the biggest mean differences between an empty board and a board with one of every piece. After manually renaming the files, we should get a *templates* directory like the picture below.

Content of *templates* directory

For fun, we can plot the heatmap for the differences between an empty board and a board with one of every piece.



Going back to recognizing the number on the piece we add the usual **threshold** over the **grayscale picture** of the number we want to recognize.

Thresholded square

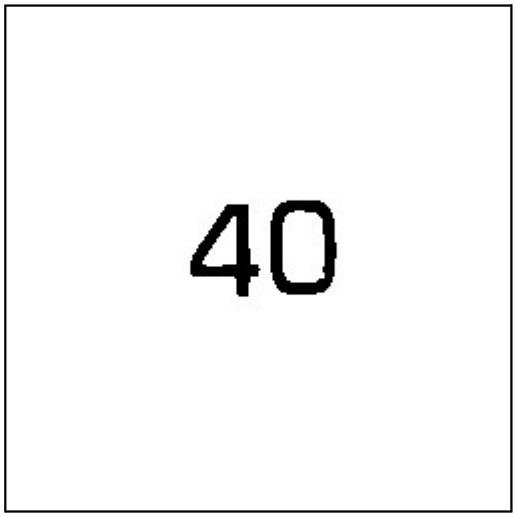
We repeat those operations on every template.

Some of the thresholded templates

Our square and the templates have **equal dimensions**, which can seem perfect at first. However, OpenCV's *matchTemplate* slides the templates over the input image and compares the template and the input image patch under it. In our case, because of the equal dimensions, the result will be a 1x1 array, which is the equivalent of **computing a distance** between the template and the input image.

We don't want to use *matchTemplate* like this since it could cause us problems because of the small **variation in orientation and position** of the number in the square (it can easily be observed if we closely look at the square and template images of the *40s* above).

This can be solved by bordering the square we want to classify so the template can be slid over the input image.

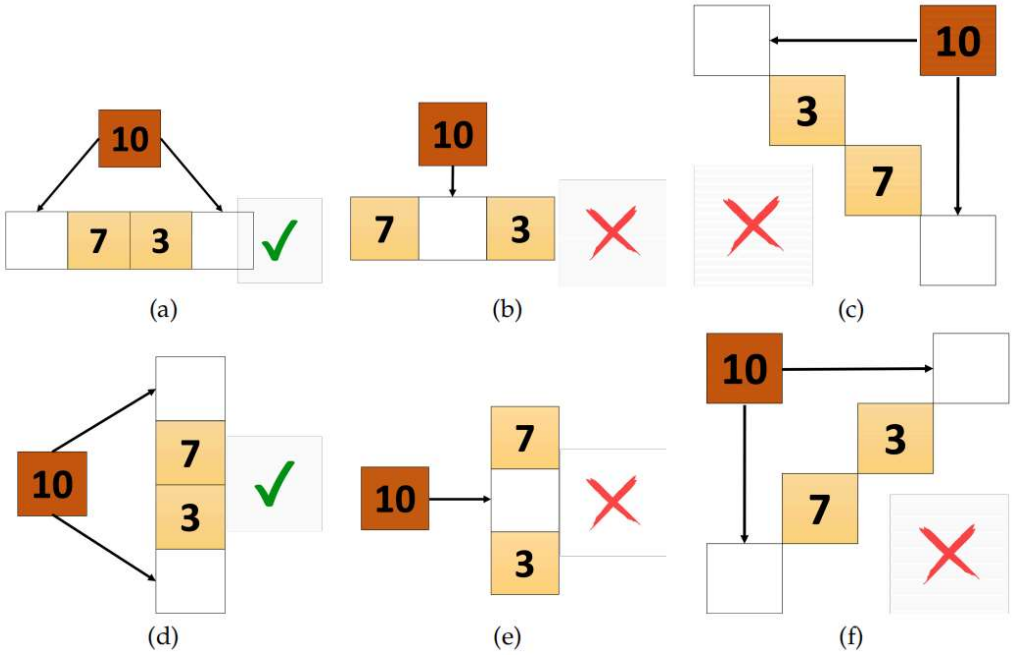


Enlarged square

Now, we can use *matchTemplate* with all of the created templates and select the one with the biggest correlation to properly recognize the number. The code for this can be found in the *classify_number* function.

iii. Implementing score computation based on the rules

I will not go into detail for the score computation since its focus is more on the **algorithmic** part, not on Computer Vision techniques. Briefly, the score is equal to the number of completed equations multiplied by 2 or 3, if the last piece placed was on a *2x* or *3x* tile, multiplied by the number of the last piece placed. The valid ways to complete an equation are in the diagram below.



Future improvements and real-world applications

As a future improvement, we can optimise *task i* by checking **only squares adjacent** to the ones already placed on the board instead of computing the differences for all the 196 squares.

This project can be used as the base for an automatic **tournament scoring system** and transform the input from a camera to graphics for a livestream feed or **help visually impaired individuals** to play Mathable by providing real-time auditory feedback on the board state